计算机网络

1. HTTP--超文本传输协议

是应用层上的一种客户端/服务端模型的通信协议，由请求和响应构成，无状态

协议：规定了双方按照约定格式才能准确的通信

无状态：两次通信间没关系

Socket 是：TCPAPI，为了方便使用 TCP/UDP 抽象出来的一层，是应用层和传输控制层之间的一组**接口**，用于组织数据去符合指定协议

WebSocket：一个 HTTP **应用层协议**--只需要一次握手然后单独建立一条 TCP 通信通道来传数据(用于传小数据不占很多宽带)

OSI 7 层模型 用处：规范地控制网络：

HTTP /FTP 文件传输协议/webSocket　TCP/UDP　IP　数据链路　物理介质

应用层 表示层 会话层 传输层 网络层 数据链路层 物理层

TCP/IP 5 层模型及功能：

1.物理层：将数据以实体呈现并传输

2.数据链路层：把数据分割成特定的可被物理层传输的帧

3.网络层：**选择传送路径**

4.传输层：给**数据编号**、控制数据流量、查错与错误处理，确保数据可靠、**按顺序、无错地传输**

5.应用层：负责在网络中的**两节点之间建立和维持通信**。文件传输

为什么分层：1.各层独立，不用知道下**层怎么实现**，调整层内工作方便

2.不会有**木桶效应**，不会因某一层**技术不完善**影响整体效率

3.易于**维护**，对每层单独调试即可

HTTP 状态码：80 未加密 明文传输 100—199 服务器收到要请求者继续执行请求的状态代码 200—299 成功处理了请求的状态代码 300-399 重定向（完成请求要其他操作） 400-499 客户端出错 500—599 服务端出错

常见状态码：100 要客户端继续发

200 服务器接收到请求，将返回结果

202 接受了请求，不确定处不处理

301 请求的链接发生永久性变化，客户端向新链接请求

302 请求的链接发生临时性变化 临时重定向

304 get 请求被运行但请求的网页没修改

400 请求无效

401 当前用户需要验证

403 服务器得到请求但是拒绝执行

404 请求失败，请求的资源没找到或者不存在

URL 构成

HTTP 请求报文：请求行(请求方法,URL,版本)，请求头(content-Type 比如 image/gif 是 gif 图片格式)，空白行(通知服务器不再有请求头)，请求体(POST 的数据)

HTTP 响应报文：响应行(版本，状态码，状态码描述)，响应头
空白行，响应体(html)
常用请求头：**Accept**(可接受响应内容类型), cache-control(指定当前请求/回复**是否用缓存**机制)

实现长连接：创建**请求方法**-设置**请求地址和参数编码格式(utf8)**-设置**长连接**-设置请**求参数**-执行
端口号：
IP 区分计算机，**端口号区分相同计算机的不同的服务，**一台计算机上可以同时提供很多个服务，如数据库服务、Web 服务。
客户端用 IP 来找服务端，用端口号来找服务端的服务，实现真正的访问

缓存机制：第一次向服务器发请求，服务器返回请求的资源，之后再要请求资源就可以用**强缓存**(200，cache-control)(从本地缓存获取数据，**不**与服务器**交互**)或**协商缓存**(304，Last-Modified)(发请求到服务器，服务器判断可不可以用本地缓存)
短连接：客户端和服务端每次 **HTTP 操作就建立一次链接，任务结束就断开，**用于 web 网站的 http 服务因为会有很多客户端连接
长连接：连接后客户端或服务端关闭，客户端再次访问时继续用这个 TCP 连接
用于操作频繁(数据库)，点对点的通讯，而且连接数要少，因为会消耗资源

HTTP 0.9 只能发 GET 请求
HTTP 1.0 支持**短连接**，而且会**请求部分对象**，但服务器把**整个对象**送过来了
HTTP 1.1 支持**长连接**，加了 **cache-control 请求头**，**宽带优化**(只送请求部分对象)
HTTP2.0:基于 **https**，具有**安全性**，使用**二进制格式**和**多路复用**(1.1 里只能完成上个请求才能发起下个请求，而 2.0 里的二进制帧可交错传输)

2. TCP 和 UDP 的区别

是传输层两个协议
1. 区别
TCP 面向连接，　　UDP 面向无连接
TCP 开销小，有拥塞控制　UDP 没有不管网络是否拥塞，客户端都能一直发
TCP 面向字节流　　UDP 基于数据包
TCP 保证数据正确性和顺序，UDP 可能丢包而且不保证顺序
组成：

UDP 很简单，除了**端口号**几乎什么都没有，**不会根据网络的情况进行拥塞控制**
变可靠: 要**接受方收到 UDP 后回复个确认包**，发送方收不到确认包就要重新发
TCP 除了有源端口和目标**端口**，还有包的**序号**(给包编号知道哪个先来)，确认序号(确认对方是否收到，解决丢包)，**状态位**(SYN 发起链接，ACK 回复,RST 重新连接，FIN 结束连接)

流量控制：
滑动窗口作用：用来**加速数据传输**，TCP 要保证“可靠”，就需要对一个数据包进行 ack 确认表示接收端收到。有了滑动窗口，接收端就可以等收到许多包后只发一个 ack 包，确认之前已经收到过的多个数据包。有了滑动窗口，发送端在发送完一个数据包后不用等待它的 ack，在滑动窗口大小内可以继续发送其他数据包
TCP **滑动窗口**：**窗口大小指一次传输几个数据**，发送方在发送时候始终保持一个窗口，只有窗口内的数据帧才被发送，同样接收方也有个接收窗口，只有在窗口内的才会被接收，通过改变窗口和窗口大小实现流量控制
窗口太大：容易**丢包**->重发->耗费带宽 **窗口太小**：**频繁**发->占**带宽**

拥塞控制(防止发送方发的太快导致过多数据注入到网络—死锁)发送端发报文速度比接收端接受速度快 拥塞：资源需求 > 可用资源
发送方有一个拥塞窗口变量，越拥塞窗口越小
知道丢包：1.定时器超时 2.收到 3 个重复的 ACK
什么时候拥塞控制：1.网络传输 TCP 报文过程中发生丢失报文 2.报文需要重传

4 个拥塞机制：
慢开始：由小到大增加拥塞窗口数值
拥塞避免：按线性增长拥塞窗口，比慢开始慢
重传定时器溢出的时候(也就是有很多报文没有按时发送到接收端或者接收端的 ACK 报文没有到达发送端)用慢开始和拥塞避免 先慢开始(指数增长)再拥塞避免(线性)
快重传：接收方收到一个失序报文段立即发出重复确认，而不是自己发数据时捎带确认
快回复：使慢开始在 TCP 连接建立和网络超时时采用
报文重传了
拥塞控制 vs 流量控制：流量控制虽然可以高效可靠的传送大量的数据，但是如果在刚开始阶段就发送大量的数据，可能会导致网络拥堵，因为网络上的计算机太多了
1.相同点

> (1) 现象都是丢包;
> (2) 实现机制都是让发送方发的慢一点，发的少一点

2.不同点

（1） 丢包位置不同
流量控制丢包位置是在接收端上
拥塞控制丢包位置是在路由器上
（2） 作用的对象不同
流量控制的对象是接收方，怕发送方发的太快，使得接收方来不及处理
拥塞控制的对象是网络，怕发送发发的太快，造成网络拥塞，使得网络来不及处理
3.联系

```
拥塞控制
    拥塞控制通常表示的是一个全局性的过程，它会涉及到网络中所有的主机、
    所有的路由器和降低网络传输性能的所有因素
流量控制
    流量控制发生在发送端和接收端之间，只是点到点之间的控制
```

应用：

UDP：DNS 协议，看视频，发语音，QQ 聊天，共享屏幕 因为效率高但准确率低
TCP：HTTP 协议，QQ 传文件，邮件，登录 因为准确率高效率低

TCP 粘包：发送方发送的几个包数据到达接收方时粘成了一包，从接收缓冲区来看，后一包数据的头紧接着前一包数据的尾

原因：发送方用 Nagle 算法—收集多个小分组，在一个确认到来时一起发送，接收方收到数据包保存到缓存里而不是直接给应用层处理，然后应用程序再到缓存读取包。如果包到缓存的速度大于应用程序从缓存读取包的速度就会造成多个包被缓存，应用程序读到多个首尾相连在一起的包。

解决：发送方关闭 Nagle 算法，应用层循环读取数据，但每条数据要有固定格式而且发送方要把数据长度一起发送，这样就能判断开始和结束的位置

UDP 不会粘包因为是面向消息传输的，有保护消息边界，接收方只接受一条独立地信息
TCP 是基于流传输的，不认为消息是一条一条的，没有保护消息边界

3. 代理

proxy 代表访问用户时是代理，比如我们访问一个网址，不希望该网站知道我们的 ip，就找了一个 proxy，网站以为 proxy 的 ip 就是我们的 ip。相反，proxy 代表被访问的服务器，则此时 proxy 是反向代理，web-server 希望对用户屏蔽一些信息就找了一个 proxy 隔在中间，此时 proxy 代表 web-server 集群，用户以为 proxy 的 ip 就是就是被访问 web-server 的 ip，反向代理的服务器要有**负载均衡**功能（在多个资源中分配负载，达到最优化资源使用，避免过载）。使用服务组件 Nginx 实现。四层负载均衡主要工作于传输层，主要是处理消息的传递而不是内容，七层负载均衡主要工作于应用层处理消息内容。七层比四层优势：可以终止网络传输，根据消息**内容**作出负载均衡决定，CPU 更密集，劣势：要为每一种应用服务专门开发一个反向代理服务器，这样限制了反向代理负载均衡的应用范围

4. TCP 三次握手

三次握手的本质是确认通信双方收发数据的能力。

第一次客户端生成一个序列号给服务端发连接请求，报文段包含了序列号和 SYN 标志位=1。服务端知道客户端发件和他自己收件能力。

第二次服务端看到 SYN=1 是个连接请求，存下客户端序列号，之后随机生成一个序列号，回复的报文里有标志位 SYN=ACK=1，新序列号，以及

ack=客户端序列端+1。客户端收到了，客户端知道客户端发件收件和服务端的发件收件能力 OK。

第三次客户端收到 SYN=1 说明同意连接，存下服务端序列号，发送的报文里有 ACK=1，ack=服务端序列号+1，序列号=第一次握手序列号+1。服务端知道服务端发件和客户端收件能力 OK

为什么不是 2 次握手？

因为要考虑连接时丢包的问题，如果只握手 2 次，第二次握手时如果服务端发给客户端的确认报文段丢失，此时服务端已经准备好了收发数据(可以理解服务端已经连接成功)，而客户端一直没收到服务端的确认报文，所以客户端就不知道服务端是否已经准备好了(可以理解为客户端未连接成功)，这种情况下客户端不会给服务端发数据，也会忽略服务端发过来的数据。如果有三次握手客户端发的确认 ack 报文丢失，服务端在一段时间内没有收到确认 ack 报文的话就会重新进行第二次握手，也就是服务端会重发 SYN 报文段，客户端收到重发的报文段后会再次给服务端发送确认 ack 报文。

5.四次挥手

四次挥手目的是关闭一个连接。比如客户端初始化的序列号 ISA=100，服务端初始化的序列号 ISA=300。TCP 连接成功后客户端总共发送了 1000 个字节的数据，服务端在客户端发 FIN 报文前总共回复了 2000 个字节的数据。

第一次挥手客户端发出连接释放，报文里有 FIN 标志位=1，序列号=1101（1 建立连接时占用得序列号）。之后客户端不能发数据只能收数据。

第二次挥手服务端收到 FIN 报文回复确认报文，里面有标志位 ACK=1，确认号 ack=1102（FIN 报文序列号+1），序列号为 2300（起始的加回复的字节数据）服务端可能还有数据没发完要处于关闭等待状态一会儿，不会立马发 FIN 报文。

第三次服务端把最后数据比如 50 字节发完了，就发送释放报文，报文里 FIN=ACK=1，ack=1102 和第二次一样，序列号=2300+50=2350。

第四次挥手客户端发确认报文，ACK=1，ack=2351，序列号=1101+1

为什么 TCP 连接有 3 次关闭时候有 4 次？

因为只有客户端和服务端都没有数据发送时才能断开 TCP。而客户端发 FIN 报文只能保证客户端没数据要发，服务端还有没有数据要发不知道。所以服务端收到客户端的 FIN 报文要先发一个确认报文等数据发完了再发 FIN 报文。所以这个不能一次性把确认报文和 FIN 报文发给客户端就多了一次

为什么客户端发出第四次挥手确认报文要 2MSL（最长报文段时长的 2 倍）之后才能释放 TCP 连接？

这样是考虑丢包问题，如果**第四次报文丢失**，服务端没收到确认号 ack 报文要重新发第三次挥手报文，这样报文一去一回时间正好是 2MSL。所以需要这么长时间确认服务端收到了

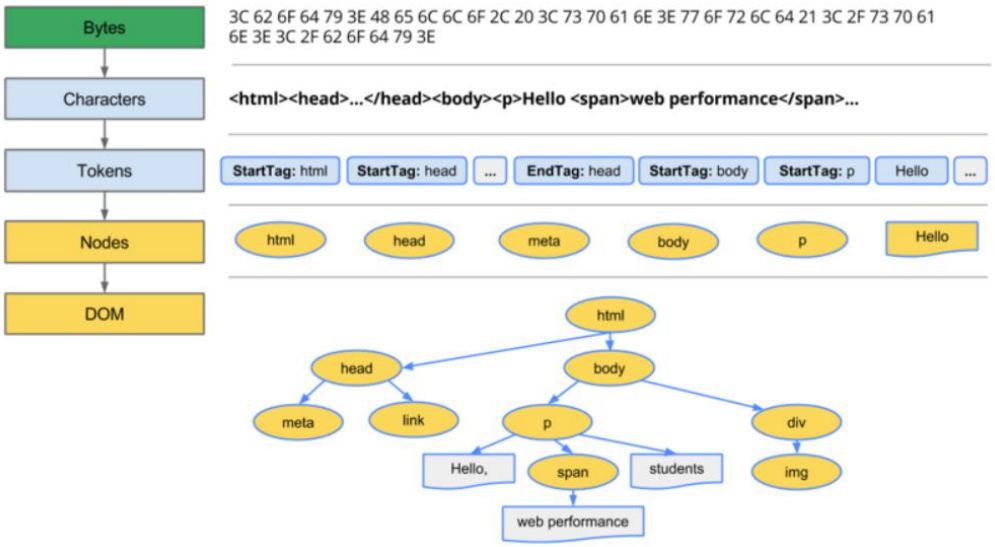
6.浏览器输入 URL 会输出什么

1.浏览器**向 DNS 发送域名**，获取 **IP**。先检查浏览器**缓存**有没有这个**网址映射**，有就调用，没有则去**操作系统的缓存**里找，没有则去找 **LDNS** 有没有，如**果 LDNS 仍然没有命中，就直接跳到 Root Server 域名服务器请求解析**

5. 根域名服务器返回给 LDNS 一个所查询域的主域名服务器（gTLD Server，国际顶尖域名服务器，如.com .cn .org 等）地址
6. 此时 LDNS 再发送请求给上一步返回的 gTLD
7. 接受请求的 gTLD 查找并返回这个域名对应的 Name Server 的地址，这个 Name Server 就是网站注册的域名服务器
8. Name Server 根据映射关系表找到目标 ip，返回给 LDNS
9. LDNS 缓存这个域名和对应的 ip
10. LDNS 把解析的结果返回给用户，用户根据 TTL 值缓存到本地系统缓存中，域名解析过程至此结束

2.得到 **IP 和端口号 80or443 之后**，会调用系统库函数 **socket**，请求一个 **TCP 流套接字，客户端向服务器发 HTTP 请求报文**：应用层：应用层发请求报文，传输层：三次握手建立连续，网络层：路由寻址,数据链路层：传输数据，物理层：物理传输 bit
3.服务器**解析请求报文，发响应报文**
4.TCP **四次挥手**，关闭连接
5.客户端**解析响应报文**，浏览器开始**显示 HTML**，js 文件开始建立 dom 树 CSS 解析器解析 css 文件得到 CSS 树，浏览器结合 CSS 数和 dom 树(下载图片等)一起建立 render 树，这里主要是排除非视觉节点，最后渲染 render 树

构建 DOM 树：



7.GET 和 POST 的区别--两种发送请求方法

HTTP 最早被用来做浏览器与服务器之间交互 HTML 和表单的**通讯协议**;后来又被广泛的扩充到**接口格式的定义**上。底层是 TCP/IP
1. 用途和幂等性
浏览器的 GET 用于获得 HTML 页面，图片，css，js 等资源。
只是读取数据 -> 可以对数据做缓存（做到浏览器 代理 服务端都可以）->幂等(副作用相同) 幂等就是多次操作副作用相同
POST 用于更新资源信息 ->不是幂等(副作用不同)->不能多次执行和不能缓存。

比如一个下单页面 ->下单成功被存到缓存，下次下单就不会对服务器发送请求->直接返回缓存的下单成功页,导致没有下单。当然 get 也可以有副作用 post 也可以没有。

2.数据位置

GET 的数据附在 **URL 之后的 queryString 传输**，更加容易被看到，更不安全

POST：请求的 **body** 传输数据，安全一些（但是 HTTP 实现**接口**发送请求时（ajax，postman 发出请求）参数可以放任何位置比如 header body querystring..这种不确定性会造成低效所以用了 REST 规定了增删改查)

RESTFUL 规范：

1.url 链接设计：采用 https 方式，有 api 关键字，有版本需要明确版本，请求链接用名词来表示资源，具体的操作方式采用请求方式来确定

2.url 响应数据设计：需要明确 状态码、错误信息、成功结果，子资源一般用子资源的接口来标注

但是私密数据在 body 里也是会被记录的，所以都不安全，所以要通过 https

3.数据大小

GET 方法传输的数据量是 **2KB** 而且在 URL 种传送的参数有**长度限制**，POST 都没有，但真正影响还是**服务器和 HTTP 的规定**

4.**请求缓存**时

GET 请求类似于**查找**过程，用户获取数据，**不用每次和数据库连接**，POST 一般是修改和删除的工作, **必须和数据库交互 ->** GET 适合请求缓存

5.GET 产生一个 TCP 数据包，POST 产生 2 个:

GET 请求，浏览器会把 http header 和 data 一起发送，服务器响应 200，

POST 请求，先发 http header,服务器响应 100continue，再发 data，服务器响应 200

但是

GET 和 POST 本质就是 **TCP 链接**，并无差别，是 HTTP 的规定和浏览器的限制导致他们不同

8.http 请求方法

GET 请求**页面信息**

POST 向指定资源**提交数据进行处理**

PUT 传送的**数据取代**指定**内容**

DELETE **删除**指定**页面**

OPTIONS 允许客户端查**看服务器性能**

HEAD 类似 GET，但返回的**响应没有内容只有报头**

TRACE **显示**服务器收到的**请求**，用于测试或诊断

9.http 和 https 的区别？

http 端口号 80，运行于 TCP 之上，**明文传输**，客户端和服务端**无法验证**对方**身份，资源消耗少，安全性低**

https 端口 443，HTTP 协议+SSL 协议，是添加了加密和认证的 http，加密通过对称加密和非对称加密混合加密 -> 消耗更多 CPU 和内存，需要花钱买**证书**(分 DV, OV, EV 三种类型 可信度越来越高，越来越严格)

为什么混合加密：

数据是被对称加密传输的，对称加密过程需要客户端的一个密钥，为了确保能把该密钥安全传输到服务器端，采用非对称加密对该密钥进行加密传输，总的来说，对数据进行对称加密，对称加密所要使用的密钥通过非对称加密传输。
过程：1.客户端向服务器发起 HTTPS 请求，连接到服务器的 443 端口
2.服务器端有一个密钥对，即公钥和私钥，是用来进行非对称加密使用的，服务器端保存着私钥，不能将其泄露，公钥可以发送给任何人。
3.服务器将自己的公钥发送给客户端。
4.客户端收到服务器端的证书之后，会对证书进行检查(看签发机构可不可信任)，如果发现证书有问题，那么 HTTPS 传输就无法继续。如果公钥合格，那么客户端会生成一个用于进行对称加密的密钥。然后用服务器的公钥对客户端密钥进行非对称加密，这样客户端密钥就变成密文了，至此，HTTPS 中的第一次 HTTP 请求结束。
5.客户端会发起 HTTPS 中的第二个 HTTP 请求，将加密之后的客户端密钥发送给服务器。
6.服务器接收到客户端发来的密文之后，会用自己的私钥对其进行非对称解密，解密之后的明文就是客户端密钥，然后用客户端密钥对数据进行对称加密，这样数据就变成了密文。
7.然后服务器将加密后的密文发送给客户端。
8.客户端收到服务器发送来的密文，用客户端密钥对其进行对称解密，得到服务器发送的数据。第二个 HTTP 请求结束

对称加密加密解密用的同一个密钥，传输过程会被截获，不安全，但传输快
其加密过程如下：**明文 + 加密算法 + 私钥 => 密文**
解密过程如下：　**密文 + 解密算法 + 私钥 => 明文**
由于对称加密的算法是公开的，所以一旦私钥被泄露，那么密文就很容易被破解，所以对称加密的缺点是密钥安全管理困难。
非对称加密用公钥加密，私钥解密，公钥无法反推导出私钥所以安全，但加密解密废时间，速度慢
被公钥加密过的密文只能被私钥解密，过程如下：
明文 + 加密算法 + 公钥 => 密文，　密文 + 解密算法 + 私钥 => 明文
被私钥加密过的密文只能被公钥解密，过程如下：
明文 + 加密算法 + 私钥 => 密文，　密文 + 解密算法 + 公钥 => 明文
SSL 是安全套接层协议
TLS 安全传输层协议

10.Session 和 Cookie

Cookie 是服务器保存在浏览器上的 key-value 格式文件，有用户信息
　　最大作用：存储 sessionId 来唯一标识用户
Session 依赖 Cookie 实现, 是**服务器分配的一块储存空间**

过程：客户端向服务器发请求，如果**服务器需要记录该用户状态**，就颁发浏览器一个 Cookie，并且自己创建一个 session -> **客户端**浏览会把 Cookie **存**起来，再请求该网站时，浏览器会把请求网址和 Cookie 一起发给服务器。服务器检查该 Cookie 来里的 sessionID 来辨认用户身份。

cookie、session(放敏感信息)与 localStorage 区别

1. 位置

cookie 数据在客户端，安全性低　session 数据在服务器，安全性高

2. 大小

Session 和 localStorage 比 cookie 大得多

3. 存在时间

localStorage 存储持久数据，只能手动删

sessionstorage 关浏览器

cookie 设置了过期时间，浏览器关了也不会删，没设置则关闭浏览器后失效

session 有失效时间，如果 session 倍访问，失效时间归 0 开始计

token

cookie 和 token 都是**首次登录**时由服务器**下发用来验证**

好处：不用账号密码 -> 减少数据库查询

区别(管理、共享)：token 由**应用管理**，可以**避开 csrf** 攻击并能在多个**服务间共享**

csrf 攻击：1.受害者登录受信任网站 A **生成 Cookie**

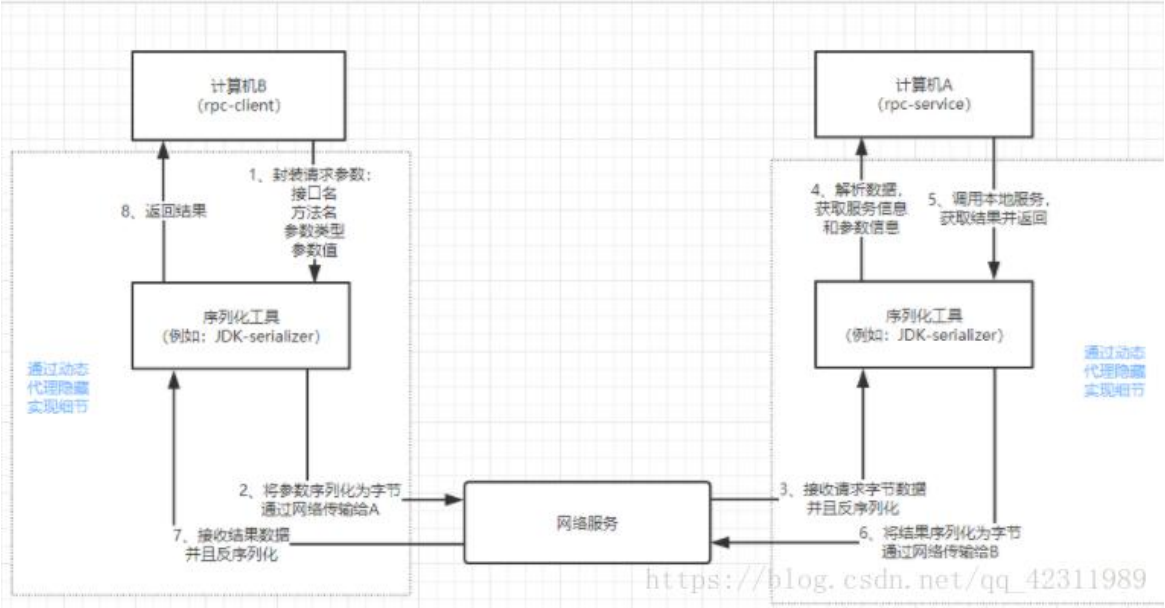
2.在不登出 A 的情况下访问危险网站 B

Unicode 16 进制　UTF-8 二进制　UTF-8 是 Unicode 的一种实现

RST 标志位表示复位，用来异常的关闭连接。用 RST 包关闭连接时不用等缓冲区的包都发出去（不像 FIN），直接丢弃缓冲区的包发送 RST 包，接收端收到 RST 包后也不用发 ACK 包确认。比如 A 向 B 连接，但 B 没有监听相应地端口，B 的 TCP 处理程序就会发 RST 包

RST 攻击：AB 建立了连接，C 伪造一个 TCP 包发给 B，让 B 丢弃缓冲区的数据而且异常地和 A 断开连接。想伪造 A 的包要知道 A 的源 IP、源端口、目标 IP、目标端口

RPC 是远程过程调用，也就是在一个服务器的应用要调用另一个服务器上的应用的方法，但不在一个内存空间，不能直接调用，只能靠网络来调用



Linux 指令

scp，管道传输：Linux 主机之间传输文件
ps -A：显示**所有**程序
ps -ef：用**标准格式**显示进程
ps aux:用 BSD 的**格式**显示进程
file：**判断**文件类型
fd：查看已被打开的文件描述符
grep walkpgdir 文件名：print 出文件中**匹配行内容**
grep -i :不区分大小写 print 出文件中匹配行
find：在目录重搜索文件，有则 print 出文件名
top：CPU 使用状态，内存使用情况
vi 替换字符串：s/vivian/sky 替换**当前**行**第一个** vivian 为 sky
s/vivian/sky/**g** 替换**当前**行**所有** vivian 为 sky
%s/vivian/sky/g 替换**所有行**所有 vivian 为 sky
more **分页展示**文件内容
less **分屏展示**内容，根据**显示需要加载内容**，对**大型文件**效率更高
head -n 5 查看文件头 5 行， tail 反之。
ifconfig:显示网络接口，子网掩码
netstat：看网络情况
netstat -anpt | grep 80：查看 80 端口 – 与网络有关用 netstat 再用 grep 找端口
Lsof -i:端口号 ：查看端口被什么进程占用
ps aux | grep sha(程序的名字)：查线程号
ps -ef |grep mysql 显示有关 mysql 的进程
查看日志：tail/head 来显示 test.log 内容 + cat -n test.log | grep “debug”查询关键字的日志
vim 进入 log 文件用/+关键字查找，下一个按 n 就行

kill -9 pid：杀死进程 + 重新加载和停止进程，发信息 9 是 SIGKILL(发给进程)

Git 指令

git branch 查看本地所有分支
git rm 文件名 删除指定文件
git log 查看 commit 日志
git clone 克隆版本库
把本地文件/文件夹 移到 github 上：
git config --global user.name “” --设置 github 用户信息
git config --global user.email “”
git init 初始化 --> 生成一个 .git 文件夹
git add . 添加当前文件夹下所有文件
git commit -m “”提交到 Repository
git remote add origin 加上仓库地址
git push origin master 将文件推到服务器上
merge 和 rebase 用于合并分支
区别：1.merge **不会**保留 merge 分支的 **commit 到日志**
2.**处理冲突(不知道要保留什么设置)**时，merge **会产生一个 commit**，rebase 不会
版本回退：
1.没用 git add.缓存代码：git checkout — filepathname
2.用了 git add：用 git reset HEAD filepathname
3.已经提交了代码：git log 看 git 提交历史，里面第一行就是 commitid，然后用这个 id 去 git reset —hard commitid 回退
4.已经 push 到远程服务器：git reset —hard
git push origin HEAD —force

操作系统

进程：一个有独立功能的程序在一个数据集上一次动态执行的过程

OS **资源分配**的最小单位,分配了内存 I/O CPU
由**程序**(描述功能)、**数据集合**、**PCB** 组成(进程描述和控制信息--标志)
每个进程有独立内存，但因为进程间上下文切换耗费大，所以发明了线程

线程：**程序执行**和 OS **调度**的最小单位 并行

协程：是一种用户态的轻量级线程，调度的话完全由用户控制，创建时不用调用 OS 功能，多个协程在一个主线程上，是异步机制 为了并发
并发的关键是你有处理多个任务的能力，不一定要同时
并行的关键是你有同时处理多个任务的能力

差别：1.**定义**

进程是操作系统资源分配和调度的最小单位，而线程是程序执行的最小单位

2. 包含

进程(有多个线程) > 线程(进程种代码执行路线) > 协程

3. **共享**

进程相互独立，线程共享内存空间，堆等，但私有栈,寄存器(程序计数器)

4.**切换**

进程切换要保存当前 CPU 环境并设置新的，线程只是保存和设置寄存器

联系：通常只有一个 CPU 并只给一个进程，总线程数>CPU 数-并行，反之并发

并发是充分利用每一个核

线程的实现方法:

1. 线程池
2. 继承 Thread 类(编写简单，直接用 this 获取线程，但不能继承其他类了)
3. 实现 Runnable 接口(重写 run()方法，没返回值，不能抛出异常)
4. callable(重写 call()方法，有返回值，可以抛出异常)

3 和 4 适合多个线程资源共享

线程通信方法：1.**全局变量**(因为内存共有) **volatile**

2.**消息队列**(因为每个**线程有自己的消息队列**)

3.**事件**

线程 5 个状态：1.NEW 新建 2.RUNNABLE(其他线程 call 了 start 方法)

3.RUNNING runnable 的线程获得时间片 4.BLOCKED 放弃了 CPU<- sleep、锁被占用 5.DEAD

start()启动线程--处于就绪状态，之后 Thread 类调用 **run()使其运行**，run()里面有线程内容，run 结束线程也终止

线程互斥：同时只能一个线程进入临界区(critical section)对资源进行读写操作

线程同步：多线程通过互斥量等东西控制线程之间执行顺序

比如一个线程对临界区写，另一个读，必须先写再读。不然结果不如预期

方法：1.互斥量 只有拥有互斥量对象的线程才能访问公共资源

2.临界区 只允许一个线程对共享资源访问

3.信号量 允许多个线程同时访问资源

线程安全：多线程访问时，采用了加锁机制，当一个线程访问某个数据时进行保护，其他线程不能进行访问直到该线程读取完，不会出现数据不一致或者数据污染

线程不安全：不提供数据访问保护，有可能多个线程先后更改数据造成所得到的数据是脏数据，不符合预期

线程池：创建多个可执行的线程放到一个池里，需要的时候不用自己创建直接从池里获取，使用完不销毁线程而是把它放回池里，线程池会在空闲时间来创建和销毁线程，这样服务器在处理用户请求的时候就不会有创建和销毁线程的开销，可以控制最大并发数

应用场景：FixedThreadPool 用于负载比较重，而且负载比较稳定的场景，比如有套负载比较重的后台系统，每分钟要执行几百个复杂的 SQL，就是用 FixedThreadPool 是最合适的。然后因为负载稳定，一般来说，不会出现突然涌入大量请求，导致 100 个线程处理不过来，然后就直接无限制的排队，然后内存溢出

CachedThreadPool：用在负载很稳定的场景的话就浪费了。因为每天大部分时候可能就是负载很低的，用少量的线程就可以满足低负载，不会给系统太大压力；但每天如果有少量的高峰期，比如说中午或者是晚上，高峰期可能需要一下子搞几百个线程出来，那 CachedThreadPool 就可以满足这个场景，然后高峰期应付过去之后，线程如果处于空闲状态超过 1 分钟，就会自动被回收，这样就避免给系统带来比较大的负载

参数：

1. corePoolSize 核心线程数，指保留的线程池大小(不超过 maximumPoolSize 值时，线程池中最多有 corePoolSize 个线程工作)
2. maximumPoolSize 指的是线程池的最大大小(线程池中最大有 corePoolSize 个线程可运行)
3. keepAliveTime 指的是空闲线程结束的超时时间(当一个线程不工作时，过 keepAliveTime 长时间将停止该线程)
4. unit 是一个枚举，表示 keepAliveTime 的单位(有 NANOSECONDS, MICROSECONDS, MILLISECONDS, SECONDS, MINUTES, HOURS, DAYS，7 个可选值) 纳秒，微秒，毫秒，秒，分，小时，天
5. workQueue 表示存放任务的队列（存放需要被线程池执行的线程队列）
6. handler 拒绝策略(添加任务失败后如何处理该任务)

拒绝策略：

1.abortpolicy(线程池默认的拒绝策略，在任务不能再提交的时候，抛出异常，及时反馈程序运行状态。如果是比较关键的业务，推荐用这个拒绝策略，这样在系统不能承载更大的并发量的时候，能够及时的通过异常发现)

2.discardPolicy(丢弃任务，但是不抛出异常。如果线程队列满了，那么后面提交的任务都会被丢弃，而且是静默丢弃)

3.discardOldestPolicy(丢弃队列最前面的任务，然后重新提交被拒绝的任务)

4.CallerRunsPolicy(由调用线程处理这个任务,如果任务被拒绝了，就让调用线程（提交任务的线程）直接执行此任务)

多线程：为了**同步完成多项任务，提高资源使用率**来提高系统效率(不是运行效率)，充分利用 CPU，更好地利用系统资源。

如果 OS 支持多个处理器，每个线程分配给一个处理器。它最有价值的地方是我们不用知道使用了几个处理器。但可能会有**共享资源的问题**，所以要用**锁**

实例：生产者消费者问题，3 个地方卖 20 张票，用同步锁保证不会卖出同一张票

单核 CPU 支持多线程，通过给每个线程**分配时间片**

进程通信/同步方式：1.无名**管道**(有亲缘关系的进程间用，数据单向流动)

2.有名管道(没亲缘关系也可以用)
管道是在内核申请一块缓冲区用于读写
3.**共享内存**(映射一段物理内存到不同进程的虚拟内存) – 最快
最快因为**不用切换到内核态**，直接从**内存里读**
不会占用地址空间，因为内核为它增加了一段
注意：因为是**临界资源**，所以要用**信号量保证原子性**
4. 消息队列(一个队列，元素是数，进程可以访问这个队列)
5.**信号量**(计数器，作为一种锁的机制管理进程对资源的访问)Linux 里要用自旋转锁
6.**套接字**(用于**不同机器**间进程通信)

虚拟内存：使应用程序**被认为有连续完整的地址空间**，实际上是在多个**物理碎片**和**磁盘**(disk)上

ThreadLocal：**保证线程安全的方法，创建一个变量**后每个线程对其访问的时候都是**访问线程自己的变量** （Java 里）子线程能获得父线程 ThreadLocal

死锁：一组进程中各个进程都**有不会释放的资源**，但因互相**申请**被其他进程**所占的资源**而处于一种**永久等待状态**

死锁 4 个必要条件：

1. 互斥条件：资源由一个进程使用，不能共享
2. 请求与保持条件：已经得到资源申请新资源
3. 非剥夺条件：已经分配的资源不能从相应进程中剥夺
4. 循环等待条件：进程组成环路，每个进程都在**等相邻进程的资源**

最根本原因：没释放一个锁要去获得另一个锁

避免死锁：破坏四个条件之一，比如 2.为获取锁时间设超时时间、只用一把锁

银行家算法：允许进程**动态地申请资源**，系统在每次实施资源分配之前，**先计算**资源分配的**安全性**，若此次资源分配安全，便将资源分配给进程（即资源分配后，系统能**按某种顺序来**为每个进程**分配资源**，使每个进程都可以顺利完成），否则**不分配**资源，让进程**等待**

sleep(): **让出 CPU** 暂停**执行**,**不释放锁**，**阻塞**线程，不考虑优先级，Tread 的静态方法

yield(): **让出 CPU** 暂停**执行**，**不释放锁**，**不阻塞线程**而是回到**就绪**状态，Tread 的静态方法

wait(): 进入**等待池(变成 blocked 状态)**，**释放锁**，只有别的线程调用 notify 才进入锁定池准备，Object 的方法，任何对象实例都能调用

join(): **等异步线程执行**完才能执行

block(): synchronized 会导致线程进入 blocked 状态

block() vs wait()：Blocked 是指线程正在等待获取锁，waiting 是指线程正在等别

的线程调用 notify，之后线程可能会进入 runnable 状态，也可能再次获得锁变成 blocked 状态
阻塞状态是线程因为某种原因放弃 CPU 使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。
阻塞的情况分三种：
等待阻塞：运行的线程执行 wait()方法，JVM 会把该线程放入等待池中。
同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则 JVM 会把该线程放入锁池中。
其他阻塞：运行的线程执行 sleep()或 join()方法，或者发出了 I/O 请求时，JVM 会把该线程置为阻塞状态。当 sleep()状态超时、join()等待线程终止或者超时、或者 I/O 处理完毕时，线程重新转入就绪状态

Shell 是一个应用程序，它连接了用户和 Linux 内核，让用户能够更加高效、安全、低成本地使用 Linux 内核
gdb：UNIX 和 UNIX-like 下的调试工具，用来运行程序，停在断点检查程序

缓存 cache 和缓冲 buffer：cache 是被磁盘读出的 buffer 是要写入磁盘的东西

cache 提高 **CPU 和内存**之间数据交换速度，把数据放在几级 cache 里，减少内存访问

buffer 提高**硬盘和内存**之间数据交换速度，把写操作集中进行，提高性能
文件描述符(file descriptor)是内存为了管理**被打开**的文件创建的**索引**(非负整数) 0 开始

IO 多路复用(同时处理多路 I/O)：有一堆文件描述符 -> 调用函数让 Kernel **监视**这些描述符，其中有描述符可以 I/O 读写操作时再**返回**，返回后我们就知道哪些文件描述符可以来 I/O 操作

3 种 Linux **阻塞时(同步)I/O 机制**：

1. select：把**文件描述符**(最多 1024 个)通过**参数传给** select，select **拷贝**到 Kernel，线程**不安全**
2. poll：相似 select 只是文件描述符能超过 1024 个，线程**不安全**
3. epoll(Linux 独有)：**只操作有变化的描述符**，和 Kernel **共享**了些**内存**来放已经可读可写的文件描述符，**减少了拷贝到 Kenel 开销**

用**事件驱动解决要遍历文件符才知道哪个可读可写**：进程只要等待在 epoll 上，epoll 代替进程去各个文件描述符上等待，哪个描述符可读可写就通知 epoll,epoll 记录下来然后唤醒进程

用户态(运行用户程序) **内核态**(运行 OS 程序)

区别：用户态比内核态**低级**，不能直接**访问 OS** 程序，占有的处理器**可被抢占**
用户态 -> 内核态 3 种情况:
1. (通常靠库函数)system call fork()
2.异常(发生异常会切换当前进程到处理此异常的内核)

3.设备中断(设备向 CPU 发出中断信号)，比如**硬盘读写完成 东西保存在 bios**

逻辑地址 -> 线性地址(虚拟) -> 通过 **MMU** 转化为物理地址

Linux 里**逻辑地址=线性地址** 因为**线性地址都是从 0 开始**
线性地址转物理地址：从进程中得到 pgdir 地址->用线性地址前 10 位找**索引**，中间 10 位找 page table 里 **page 位置** ->加上后 12 位得到**物理页**位置

Linux 3 种**内存模型**：

1. FLAT：访问**物理内存**时候，地址空间连续，不空洞，会浪费
2.Discontinuous： 地址空间不连续，有空洞
3.Sparse：连续的地址空间被**分段**，每段都支持**热插拔**

最佳置换算法 OPT 性能最好，但无法实现。先进先出 FIFO 简单，但性能差。最近最久未使用 LRU 性能好，是最接近 OPT 算法性能的，但是需要专门的硬件支持，开销大。

时钟置换算法 CLOCK 是一种性能和开销均平衡的算法

CLOCK 算法思想：为每个页面设置一个**访问位**，再将内存中的页面都通过指针连成一个循环队列。当某个页被访问时，其访问位置 1.当需要淘汰一个页面时，只需检查页的访问位。是 **0 就换出**；是 **1 就把访问位改为 0**，继续检查**下一个**页面
若**第一轮扫描**中所有的页面**都是 1**，则将这些页面的访问位**都置为 0 后**，再进行**第二轮扫描**（第二轮扫描中一定会有访问位为 0 的页面，因此简单的 CLOCK 算法选择一个淘汰页面最多会经过两轮扫描）。

要用 Wait/waitpid:取得子进程运行状态，因为父进程和子进程是个**异步**过程(父进程不知道子进程什么时候结束)
孤儿进程：父进程退出，它的子进程还在运行->由 init 收集它状态 没有危害
僵尸进程：子进程退出，但父进程没用 wait 获得子进程状态，子进程的进程描述符仍在系统中 ->多了导致没用可用进程号->不能产生新进程

静态库(扩展名.a/.lib)**compile 时直接整合到程序**，优点：compile 成功的文件可以**独立运行**，但函数库升级要重新 compile
动态库(.so/.dll) 文件需要用库函数才读取库函数(可执行文件无法独立运行)，优点：节省内存并**减少页面交换**，实现进程资源共享

OS 内存管理

a)最先适配：分配n个字节，查找并使用第一个可用的大于等于n个字节的空闲块。

b)最佳适配：查找并使用不小于n的最小空闲块。

c)最差适配：查找并使用不小于n的最大空闲块。

d)快速适配：为常用大小的空闲区建立单独维护的链表。例如：假设512B,1KB,4KB,10KB……为经常需要的大小的内存块，为这些内存块，建立单独的链表。进程申请内存时，通过该链表，直接使用这些空闲的内存块。

	优点	缺点
最先适配	简单 在高地址空间会留有较大块的空闲分区	在分配大块空闲块时，基本要遍历整个链表，所以时间会比较慢
最佳适配	可以最大程度的匹配进程需要的内存 避免大的空闲分区被拆分 恰好适配时，很好的减少外部碎片	释放分区慢 容易产生很多无用的小碎片
最差适配	避免出现小而多的空闲分区	释放分区较慢 容易破坏大的空闲分区
快速适配	可以快速的寻找指定大小的空闲分区	释放分区慢，临近空闲分区合并费时。

Python(一切都可看做对象)

1. 去首尾空格：strip，去掉左空格：lstrip，去掉右空格：rstrip
2. list：有序集合，靠索引增删 insert，pop(index),append 和 pop()是对 list 末尾增减。
3. tuple：有序列表，只能用[]获取元素，只能用[][]改 tuple 里的 list 的元素
4. 切片 startindex：endindex：step，
   start 省略时，step 为正则从起点开始，为负 从终点开始
   end 省略时，step 为正时取到终点，为负时取到起点
   取偶数位[::2]取奇数位[1::2]

5. 浅拷贝：只拷贝数据集合最外边的一层，深层的数据只是做了内存地址引用，并没有拷贝。假如 a 是一个包含数组的数组，b 浅拷贝了 a，则 a 改了他的数组元素里的数组元素，b 也会相应地改。
深拷贝：完全拷贝数据集合的所有数据，与源数据再无相关。

浅拷贝

- 只拷贝数据集合最外边的一层，深层的数据只是做了内存地址引用，并没有拷贝。

例如一个列表里面嵌套了另一个列表：[1, 2, 3, [4, 5]]

```
>>>import copy                # 导入模块
>>>a = [1, 2, 3, [4, 5]]
>>>b = copy.copy(a)          # 使用浅拷贝拷贝
>>>b                         # 打印输出b
[1, 2, 3, [4, 5]]
>>>id(a),id(b)               # 打印输出a、b
(2692110485192, 2692110468616)
>>>a is b                    # a和b的内存地
False
>>>a[3] is idb[3]            # 内层的b的[4,
True

>>>a[3][1] = 10              # 更改a的内层列
>>>a                         # 打印输出a
[1, 2, 3, [4, 10]]
>>>b                         # 打印输出b，可
[1, 2, 3, [4, 10]]
```

6. 常用库：numpy(科学计算包)　beautifulsoap(爬虫相关)
　　requests (HTTP 请求库)　　pip(安装和管理 python 包的工具)
7. 在运行时才确定对象的类型和内存--动态类型　(不像 Java，C++)
8. 内存管理：用一个私有 Heap 来放所有对象和数据结构，我们无法访问
　引用计数：x=3.14,x 是第一个引用,y=x，y 是第二个引用，计数=2
　垃圾回收：清楚引用计数=0 的对象的内存和互相引用的两个对象
　用 id 得到对象身份标识，is 比较对象地址
9. 数据类型：不可变：Number，String，tuple
　　可变：List，dictionary，set
　a=1 b=a b+=1 a 还是 1
　不可变是指 b=a 会分配 b 一个新的内存，并把 a 的值复制给 b
　可变指=后两个对象指向同一个内存
10. 内存管理：创建大量消耗小内存的对象，频繁调用 malloc 会导致很多内存碎片，减低效率，内存池就是先在内存申请一定数量、大小相等的内存块备用，有内存需求先从内存池分配，不够再申请，这样就能减少内存碎片，提升效率
小于 256bits，pymalloc 申请内存
大于 256bits，new/alloc 申请内存
回收：对象引用计数=0，从内存池来的到内存池去

读文件的过程 1、进程调用库函数向内核发起读文件请求；
2、内核通过检查进程的文件描述符定位到虚拟文件系统的已打开文件列表表项；

3、调用该文件可用的系统调用函数 read()

3、read()函数通过文件表项链接到目录项模块，根据传入的文件路径，在目录项模块中检索，找到该文件的 inode；

4、在 inode 中，通过文件内容偏移量计算出要读取的页；

5、通过 inode 找到文件对应的 address_space；

6、在 address_space 中访问该文件的页缓存树，查找对应的页缓存结点：

1.如果页缓存命中，那么直接返回文件内容；

2.如果页缓存缺失，那么产生一个页缺失异常，创建一个页缓存页，同时通过 inode 找到文件该页的磁盘地址，读取相应的页填充该缓存页；重新进行第 6 步查找页缓存

写文件

前 5 步和读文件一致，在 address_space 中查询对应页的页缓存是否存在：

6、如果页缓存命中，直接把文件内容修改更新在页缓存的页中。写文件就结束了。这时候文件修改位于页缓存，并没有写回到磁盘文件中去。

7、如果页缓存缺失，那么产生一个页缺失异常，创建一个页缓存页，同时通过 inode 找到文件该页的磁盘地址，读取相应的页填充该缓存页。此时缓存页命中，进行第 6 步。

8、一个页缓存中的页如果被修改，那么会被标记成脏页。脏页需要写回到磁盘中的文件块。有两种方式可以把脏页写回磁盘：

1. 手动调用 sync()或者 fsync()系统调用把脏页写回

2. pdflush 进程会定时把脏页写回到磁盘

同时注意，脏页不能被置换出内存，如果脏页正在被写回，那么会被设置写回标记，这时候该页就被上锁，其他写请求被阻塞直到锁释放

Java

创建对象的方法：1.new

2.通过 class 类的 newInstance class.newInstance()

3.通过 Constructor 的 newInstance

4.使用 clone 方法，需要实现 Cloneable 接口

JVM 里 new 对象时,堆会发生抢占吗?怎么去设计 JVM 的堆的线程安全

会，假设 JVM 虚拟机上，堆内存都是规整的。堆内存被一个指针分为两部分，左边塞满了对象，右边是没使用的区域。每一次 new 对象，指针就会向右移动一个对象 size 的距离。如果我们多线程执行 new 对象的方法，一个线程在给一个对象分配内存,指针还没有来的及修改,其它线程给另一个对象分配内存,而且这个线程还是引用之前的指针指向，这样就会出现抢占，就是指针碰撞

解决：在 JVM 新生代开辟一块 TLAB(线程本地分配缓存区)，也就是一块线程私有的内存分配区域。然后用这块区域放小对象，因为 Java 里有很多小对象而且小对象都是用完就被垃圾收集的，也不会被线程间去共享，而且因为线程私有所以对象分配的时候不用锁住整个堆，不存在竞争的情况，直接在自己缓冲区分配就行

TLAB **缺点**：

但 TLAB 只是让每个线程有私有的分配指针，底下存对象的内存空间还是

给所有线程访问的，只是其它线程无法在这个区域分配而已 +大小固定

异常处理机制

分为**抛出异常**和**捕捉异常** 异常分 **Error** 和 **Exception**(**受检异常**：用 **try catch**
非受检异常：**运行**时错误，比如/**0**)

多线程异常处理：线程不允许抛出没捕获的 checked exception—线程要把自己 exception 在自己的 run 方法里用 try catch 处理掉，异常被 Thread.run()抛出后不能在程序捕获，只能靠 jvm 捕获。JVM 会调用 dispatchUncaughtException 方法来找异常处理器，再调用他的 uncaughtException()方法来处理异常的，并且是直接打印到控制台

Java 里是**值传递**，C 里是引用传递
值传递：调用函数时，将实际参数(地址)复制一份到函数中，函数对参数修改不会影响实际参数
引用传递：传实际参数的地址到函数，对参数修改会影响实际参数

空指针发生的情况：
1.访问或修改 null 对象的字段。
2.如果一个数组为 null，用属性 length 获得其长度时，访问或修改其中某个元素
3.在需要抛出一个异常对象，而该对象为 null 时
4. 先判断是不是 null 再判断 equal(“”) 不然会报错 NullPointer

实现 **Serializable** 接口实现序列化(程序不运行仍保存其信息)—把对象以字节序列的形式保存
反序列化：通过字节序列得到原对象。用于系统间通信

泛型：通过参数化类型来实现在同一份代码上操作多种数据类型

类型擦除：List<Object>和 List<String> compile 后都会变成 List
泛型擦除：

- 若泛型类型没有指定具体类型，用Object作为原始类型；
- 若有限定类型< T exnteds XClass >，使用XClass作为原始类型；
- 若有多个限定< T exnteds XClass1 & XClass2 >，使用第一个边界类型XClass1作为原始类型；

4.0-3.6=0.400000001 因为 **2 进制的小数无法精确地表达 10 进制小数**

接口与抽象类

都用 abstract 声明，有抽象方法的类一定要声明成抽象类
抽象类不能被实例化(类名 a = new 类名())，只能被继承
区别：

抽象类 (is-A 关系)	接口 (like-A 关系)
有方法实现和构造器	Java8 之后有方法实现，无构造器
子类不是抽象类要提供所有方法实现	子类要提供所有方法实现

方法可以有 public protected　　方法只能 public
一个类能实现多个接口但不能继承多个类 1

集合类

1.来自 **Collection**(List,Set,Queue)
List：按位置存储数据的对象，有序 LinkedList
Queue：PriorityQueue 为元素提供优先级
Set：和 List 区别是不允许重复元素
除了 Map 类都 iterable
2.来自 **Map**(TreeMap--基于红黑树实现)
查找效率是 O(1), HashMap, LinkedHashMap(保证顺序)

重写和重载 Override 和 Overload

重写：子类实现父类方法声明一样的方法
子类方法访问权限大于父类(父 protected，子 public)，
子类返回类型和抛出异常类型和父类一样或为其子类型
(父 List<Integer>子 ArrayList<Integer>)
static(静态)方法只能被继承不能被重写
重载：同一个类中两个方法名字一样，但是参数类型、个数、顺序至少有一个不同

面向对象(OOP)是一种思想 三大特性

1.封装：将类的某些信息只能类内部访问，public 封装性最差，this
2.继承：子类拥有父类所有属性和方法, 除了 private
static(静态)方法只能被继承不能被重写,，final 不被继承和重写
3.多态：对象的多种形态--必要条件：继承，重写，向上转型
实现多态：继承和接口
1.引用多态:父类引用指向本类和子类对象：Animal obj1 = new Dog();子类
2.方法多态:创建子类对象，调用子类重写的方法或继承的方法

拷贝

clone(**浅拷贝**)拷贝基本类型值 深拷贝要实现 cloneable 接口并重写 clone 方法
引用拷贝：Object obj = new Object()　Person cur = obj

String, StringBuffer, StringBuilder

3 者都是用 char 数组存字符串
String 是不可变类，对 String 进行改变会生成新的 String 对象，然后指针指向新生成的 String，浪费内存
StringBuffer 和 StringBuilder 的对象被修改不会产生新的对象，所以字符串经常改变的话用它们
StringBuffer 线程安全因为所有公开**方法**都是用 **synchronized 修饰**的但因为要加锁所以性能差 速度慢
每次获取 toString()都会直接用缓存区 toStringCache()值来构造一个字符串
StringBuilder 不是线程安全的，不能被同步访问，但速度快

需要复制一个字符串数组再构造一个字符串
一般用 StringBuilder，除非要考虑线程安全

JVM

JVM 内存组成：1.计数器(当前线程执行的字节码的行号指示器，(PC)存指向下条命令的地址，切换线程后知道要从哪开始执行，线程私有) 2.虚拟机栈(方法执行的模型，线程私有) 3.本地方法栈(线程私有) 4.堆(共享) 5.方法区(一块线程共享内存)

JVM 内存模型决定一个线程对共享变量的写入何时对另一个线程可见

把线程间共享变量放在主内存，每个线程都有一个私有的本地内存

JVM 栈(放引用)：

1. **存**局部变量和方法调用，FIFO 数据结构，
2. **空间**比 Heap 小，创建和释放存储空间快，因为只要移动栈顶指针
3. 栈内存属于单个线程因为线程**不共享栈**

JVM 堆 – 被分成新生代和老年代

1. **存**储实例化对象和数组
2. **分配内存花的时间久**
3. 内存中所有对象**对线程可**见

垃圾回收---GC　垃圾：　无引用的对象

JVM 提供的在空闲时间不定时**回收无对象引用的对象占据的内存空间的机制**
通知 GC 回收：1.System.gc　2.把不用的对象赋值 null
引用类型：存的值是另一块内存的起始地址

强引用：只要引用存在就不回收 Object obj = new Object();
软引用：系统**内存不够**时回收，描述**可能有用**的对象
弱引用：**GC 工作时**就被回收
虚引用：**无法取得对象实例**，只是希望被回收时**收到系统通知**

解决循环引用(只用强引用引起，导致对象永远不能被释放)：如果 A 强引用 B，那 B 引用 A 时就用弱引用，判断是否为无用对象时仅考虑强引用计数是否为 0，不用关心弱引用数

判断垃圾算法：1.**引用计数算法**(引用->计数器+1，引用释放-1，**0 时清除**)

2.**根搜索法/可达性分析法**(从 **GC Root(虚拟机栈、静态变量、线程)**开始**往下**搜索，**清除没引用链的对象**)

回收算法：1.标记清除法(**标出所有**要回收的对象统一回收)--效率低

2.标记整理法(**标出**要回收的对象，让存活的向一端**移动**，直接清边界外的)-无
3.复制法(把内存**分成两块**，先用一块，把**存活的对象放另一块**)-占内存
4.分代收集法(把内存分成**新生代**和**老年代**，分别用**复制法**和**标记整理法**)
新生代分 3 个区域 Eden，from Survivor，To Survivor—这样分时为了更好地管理堆内存里的对象，包括内存的分配和回收

堆的内存模型大致为：

不分代的话得所有 root(一组活跃的引用)要扫一遍然后要知道是不是在引用链上还要扫一遍

对于新生代，对象存活期短，所以用复制法，只用关心哪些要被复制，只用标记和复制很少的存活对象，不用遍历整个堆，因为大部分是要丢弃的，缺点是浪费一半内存

对于老年代对象特点用标记-清理法，这部分如果用复制法的话没有额外空间担保(因为新生代放不下会放到老年代)，而且对象存活率高，复制的开销大

如果不分代的话，因为老年期的对象长期存活，总的 gc 频率和分代以后的年轻代 gc 频率差不多，但每次都要从 gc roots 完整来堆遍历，大大增加了开销

内存泄漏：非必要的对象引用**没清除**(数组里**添加对象后不处理**)，**未释放的资源**

内存泄漏的情况：

1.静态集合类如 hashmap、Linkedlist，如果这些容器是静态的，它们的生命周期与程序一致，容器的对象在程序结束前不能被释放。

2.改变对象哈希值导致 hashset 中无法删除当前对象

3.一个变量的定义的作用范围大于使用范围

4.内部类持有外部类

5.各种连接，比如数据库连接，没用 close 方法就不会去回收了

智能指针：一种抽象的**数据类型**，会**跟踪**指向它们的**对象进行内存管理**，通过使资源自动分配来**防止内存泄漏**。指向对象的指针被破坏时(超出范围)，对象**也会被破坏**

反射：Java compile 后会生成.class 文件，反射就是通过**字节码文件**找到类提供**类运行时的信息**。借助 **field**(可以用 get 和 set 得到和修改 field 对象) **method**，**constructor**，**class** 这 4 个类。　　缺点：性能消耗大

反射实现原理：在 JVM 层面，java 的**对象引用**不仅要可以**直接**或间接地得到**对象类型**，更应该可以根**据索引**能得到对象类型

JVM 加载.class 文件：类加载器将类的.class 文件中的二进制数据读入内存，将其放在运行时数据区的方法区内，然后再 heap 创建 java.lang.Class 对象用来封装类在方法区的数据结构

双亲委派：类加载器**加载.class 文件**时，先**递归**地把这个任务委托给上级类加载器，上级类加载器没加载，自己才加载

作用：**防止重复加载**一个.class 文件，保证核心.class 文件不被篡改

线程同步的实现/机制

1.Synchronized(独占锁, 可以修饰方法和代码块)

原子性：两个线程无法同一时间操作数据(要连续完成) + 线程访问的同步有序性

底层原理：线程 1 要加锁，执行 monitorenter 指令->看计数器 -> 0 就**获取锁并计数器改为 1** -> 线程 2 想加锁执行了 monitorenter 指令发现计数器是 1 就阻塞->线程 1 执行完释放锁，执行 monitorexit 将计数器置为 0->线程 2 进入

2. **Lock** 保证多个线程都是读或写时候就能一起进入 (CAS(保证了原子性—串行)和 volatile 实现) 保证一个线程有资源，其他 **spin**

ReentrantLock(重入锁)实现**线程安全**: 底层通过 **CAS 操作和 AQS 队列**去维护 state 变量的状态

流程：先通过 CAS 操作尝试修改 state 状态获取锁，如果获取失败就判断当前占用锁的是不是自身，如果是的话就进行重入，如果不是就进入 AQS 队列等待

3. **volatile**(线程同步的轻量级体现，但只能用于**变量**)

可见性：变量被 volatile 修饰后一个线程修改这个变量值其他线程可见

有序性：代码执行顺序就是语句顺序(不会发生指令重排(为了效率))

底层原理：修饰后在总线开启 MESI **缓存协议**和 CPU **总线的监听** -> 这个变量值**修改就会**传到总线 -> 总线**告诉其他**在使用这个变量的线程 -> 其他线程**获取新**的值

4.**wait()/notify()**

volatile 比 Synchronized 性能好

Lock 有 Synchronized 所有功能, 但需要手动加锁/解锁(因为可以手动控制，所以再复杂的并发场景用得上), Synchronized 会加锁解锁，操作简单，用于一般并发场景

ReetrantLock 和 synchronized 的区别

从底层实现上来说，synchronized 是 JVM 层面的锁，是 Java 关键字，可以修饰方法和代码，它是通过 monitor 对象来完成，对象只有在同步块或同步方法中才能调用 wait/notify 方法

ReentrantLock 是从 jdk1.5（java.util.concurrent.locks.Lock）提供的 API 层面的锁

synchronized 的实现涉及到锁的升级 (具体为无锁、偏向锁、自旋锁、向 OS 申请重量级锁)

ReentrantLock 实现是通过利用 CAS（CompareAndSwap）自旋机制保证线程操作的原子性和 volatile 保证数据可见性来实现锁

2.能不能手动释放：

synchronized 不用用户释放锁，它的代码执行完系统会自动让线程释放锁

ReentrantLock 要手动释放锁，没释放可能会导致死锁，在 final 语句块里解锁

同一个线程在外层方法获取锁的时候，进入内层方法会自动获取锁。好处：避免死锁

3.能中断：

synchronized 是不可中断类型的锁(除非加锁的代码中出现异常或正常执行完成)

ReentrantLock 可以中断，(可以通过 trylock(long timeout,TimeUnit unit)设置超时方法或者将 lockInterruptibly()放到代码块中，调用 interrupt 方法进行中断）

4.公平锁？

synchronized 是非公平锁

ReentrantLock 可以是公平锁也可以是非公平锁(通过构造方法 new ReentrantLock 时传入 boolean 值进行选择，为空默认 false 非公平锁，true 为公平锁)

5.能不能绑定 condition：

synchronized 不能绑定

ReentrantLock 可以绑定(通过绑定 Condition 然后结合 await()/singal()方法实现线程的精确唤醒，而不是像 synchronized 通过 Object 类的 wait()/notify()/notifyAll()方法要么随机唤醒一个线程要么唤醒全部线程)

6.锁的对象

synchronzied 锁的是对象，(锁保存在对象头里面，根据对象头数据来标识是否有线程获得锁/争抢锁)

ReentrantLock 锁的是线程，(根据进入的线程和 state 标识锁的获得和争抢)

AQS（Abstract Queued Synchronizer）

AQS 是一个用来构建锁和同步器的框架，可以构建比如 ReentrantLock，信号量，ReentrantReadWriteLock

核心思想：如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，同时把共享资源设置为锁定状态。

如果被请求的共享资源被占用，那就要一套线程阻塞等待和被唤醒时锁分配的机制，这个机制 AQS 是用 CLH 队列锁实现的，就是把暂时获取不到锁的线程加入到队列里。然后 AQS 使用一个 voliate int 成员变量来表示同步状态，它是通过内置的 FIFO 队列对获取的线程排队

AQS 用 CAS 对这个同步状态进行原子操作来实现对它的值的修改。

AQS 两种资源获取方式：独占式(只有一个线程有锁，又根据是否按**队列的顺序**分为公平锁和非公平锁，如 ReentrantLock)和共享式(多个线程同时获取锁访问资源，如 Semaphore/CountDownLatch，Semaphore、CountDownLatCh、CyclicBarrier)
ReentrantReadWriteLock 可以看成是组合式，允许多个线程同时对某一资源进行读

AQS 底层：用了模板方法模式， 自定义同步器了，然后在实现的时候只要实现共享资源 state 的获取与释放方式就可以了 (至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS 已经在上层已经帮我们实现好了)

CLH 同步队列是怎么实现非公平和公平的？--公平锁和非公平锁

CLH(FIFO 的双向链表)用于等待资源释放的队列，AQS 用它来管理同步状态

公平锁和非公平锁在于 hasQueuedPredecessors()方法，如果头节点不是尾节点，第一个节点不为空，而且当前节点是头节点就返回 true

线程在 doAcquire 方法里获取锁的时候会先加入到同步队列，之后再根据情况再陷入阻塞

阻塞后的节点一段时间后醒来，这个时候来了更多的新线程来抢锁，这些新线程还没有被加到同步队列里去，也就是还在 tryAcquire 方法里获取锁

在**公平锁**情况下，这些新线程会发现同步队列里有节点在等待，然后这些新线程就不能获取到锁，就去排队了

在**非公平锁**下，这些新线程会跟排队苏醒的线程抢锁，失败的线程就去同步队列里排队。所以这个**公不公平针对的是**苏醒线程和还没加到同步队列的线程，

然后根据 AQS 节点唤醒机制和同步队列的 FIFO 性质，那些已经在同步队列里阻塞的线程，它们内部其实是公平的，因为它们是会按顺序被唤醒的

Java 锁的类型

1.乐观锁：拿数据时认为别人不会修改，所以不会上锁，但是在更新时候会
适用于读操作多时，实现方式有 compare and swap(CAS)算法
CAS 是线程安全的，因为他有原子性(全执行成功 or 全失败)
悲观锁：每次拿数据都认为别人会改，所以每次都上锁
适应于写操作多的场景，因为加锁保证写时数据正确
synchronized 和 lock 都是悲观锁

2.独享锁(一次只能一个线程持有) 共享锁(可以被多个线程持有)
实现：互斥锁和读写锁

3.公平锁：线程按申请锁的顺序获取锁，非公平锁相反(都是双向链表)
非公平锁优点：吞吐量比公平锁大

4.可重入锁：同一个线程在外层方法获取锁的时候，进入内层方法会自动获取锁。好处：避免死锁

Java IO

针对被调用者：
同步：发起调用 ->被调用者处理完请求再返回(调用者等待)
异步：调用 -> 被调用者立马回应收到(没返回结果)->调用者处理其他请求->被调用者依靠回调、事件等机制返回结果
针对调用者：
阻塞：发起请求，调用者一直等请求结果返回
非阻塞：发起请求，调用者不用一直等，可以去干别的事情
Java 对 OS 的 IO 模型的封装：

1.BIO(Blocking IO):同步阻塞模型，数据读写阻塞在一个线程内完成，应用于连接少，低负载和并发，JDK1.4 之前

2.NIO(Non-Blocking IO):同步非阻塞模型，读写数据到缓冲区，应用于连接多且短，高负载和并发，比如聊天服务器，JDK1.4 开始有

3.AIO(Asynchronous IO)：异步非阻塞模型，基于事件和回调实现，也就是操作后直接返回，不会堵塞在那，当后台处理完成，OS 通知响应线程进行后序操作, 应用于连接多且长，比如相册服务器，JDK7 开始有

Object 类的方法：equals() hashCode() wait()

equals 重写场景：比较对象内容 Sting，Integer -> 使 equals 和==不同

hashcode 重写场景：不希望造成多个对象 hash 值相同，值不被覆盖

相等的对象必须有相等的哈希码，两个对象 hashCode 相同他们不一定相同

Hashmap 和 Hashtable 区别

底层都是哈希算法，接口一样

HashMap：区别：1.线程不安全(多线程下 put 会形成环导致死循环)因为 1.7 里扩容会造成数据丢失 1.8 会有数据覆盖 2.可以存 null:null 3.继承 AbstractMap 类

put 原理：先对 key 用 hashcode 方法，然后用 hashcode 的结果找到 bucket 位置来存键值对，也就是作为了 Map.Entry，如果 hashcode 一样就用 equals 比较，因为 hashcode 一样不一定就相同，还一样就替换

get 原理：对 key 用 hashcode 后得到桶位置，之后用 equals 找链表里的节点

作为 key：String，Integer 这样的 wrapper 类，因为他们是不可变—线程安全，也是 final 的而且重写了 equals 和 hashcode 方法

1.7：版本当哈希冲突严重时，桶上的链表会很长，查询很慢 O(n)

1.8：链表长度小于 8，用链表 因为查询快

链表长度 8+数组长度大于 64 转化为红黑树--泊松分布统计得出

查询为 O(logn) 不用 AVL 树因为插入太慢，经常会调整树的结构

扩容：初始大小(固定 2 的倍数)=16 扩容因子=0.75 扩容一倍 当前 12 容量 16 再扩容。0.75 是提高空间利用率和减少查询成本的折中，主要是泊松分布，0.75 的话碰撞最少大小。

扩容机制：单线程调用 rehash 把链表遍历，把元素每次用 transfer 方法放到新的链表头，链表元素次序会反过来因为每次都是从头插入来避免 O(n)尾部遍历

多线程下 transfer 会导致**死锁**，比如 A.next=B, B.next=A— 用 ConcurrentHashMap

HashTable：区别：线程安全(每个方法已有 synchronize 方法，所以多并发时候用)， key 或 value 不能为 null，继承 Dictionary 类(废弃了)，用 hashcode 取余计算 hash 值

HashSet 实现了 Set 接口，是 HashMap 的一个实例，只存 key 不存 value，因此重写了 equals 和 hashCode 来判断 Key 存不存在(先用 hashcode 和其他对象比较结果，一样再用 equals，返回 true 说明有一样的存在)，不存在才插入，所以 key **不重复**

ConcurrentHashMap

1.7(两个静态内部类)(底层是分段数组和链表)用 HashEntry(封装映射表的键值对)和 Segment(充当锁)守护若干个桶(每次在每个链表头插入节点，所以节点顺序和插入顺序相反)每个线程只能访问不同的段，提高了并发率

线程安全，因为它用了分段锁，把一个 HashMap 切割成 Segment 数组，然后 Segment 可以看成一个 HashMap，然后 Segment 继承了 ReentrantLock，在操作的时候会给 Segment 一个对象锁，想要修改 HashEntry 必须获得对应的 Segment

的锁　但是容易冲突让链表太长，这样查询就慢了
1.8(底层和 HashMap 一样链表长度大于 8 变成红黑树)放弃了 segment，在节点上用 **CAS+Synchonized**(锁住红黑树的 root)来保证**并发的安全**，只要 hash 不冲突就不会有并发问题。然后是用 volatile 关键字来记录元素个数，这样变量值变了的话其他线程也能看见就不会脏读了，提升了可见性
优化/为什么用：
HashMap 用 synchronized 修饰，对象整体会被锁住了
HashTable 大小增加到一定的时候，性能会急剧下降，因为迭代时候要被锁很久

ArrayList、LinkedList

1.Array 能**包含基本类型和对象类型 + 大小固定**
2.ArrayList 只能包含**对象类型+大小动态变化** 访问效率高, 插入和删除可能要移动整个 list。**初始大小=10 每次 1.5 倍扩容**
Add 底层：add(元素)会加在最后，不会触发底层的数组的复制，add(元素,index)会触发数组复制，最坏时间复杂度是 O(n)。new ArrayList 得时候是一个空的 object 数组，大小=0，第一次加元素，大小变成 10，每次 add 都会去检查数组空间够不够，不够就用 grow 方法扩容 1.5 倍，扩容完用 System.arraycopy 拷贝数组
Remove 底层：删除一个元素后把后面的元素往前移一个 index 并且把最后一个 index 设为 null，底层移动是用 System.arraycopy(elementData, index + 1, elementData, index,
numMoved);把后面的元素到前一位
3.LinkedList 增删节点效率高，只是修改了引用地址所以是 O(1)，但链表定位到要增删的地方是 O(n)，但写入是 O(1)，Arraylist 是 O(n)。所以 Arraylist 擅长读取，链表擅长写入　　双向链表　没有扩容机制
Add 底层：

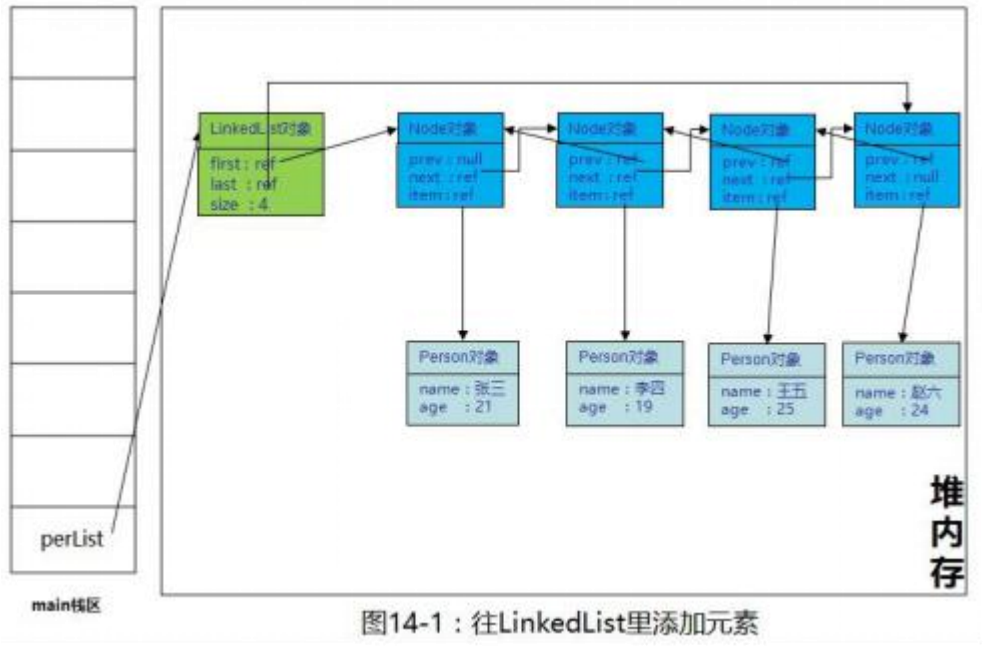


图14-1：往LinkedList里添加元素

Remove 底层：

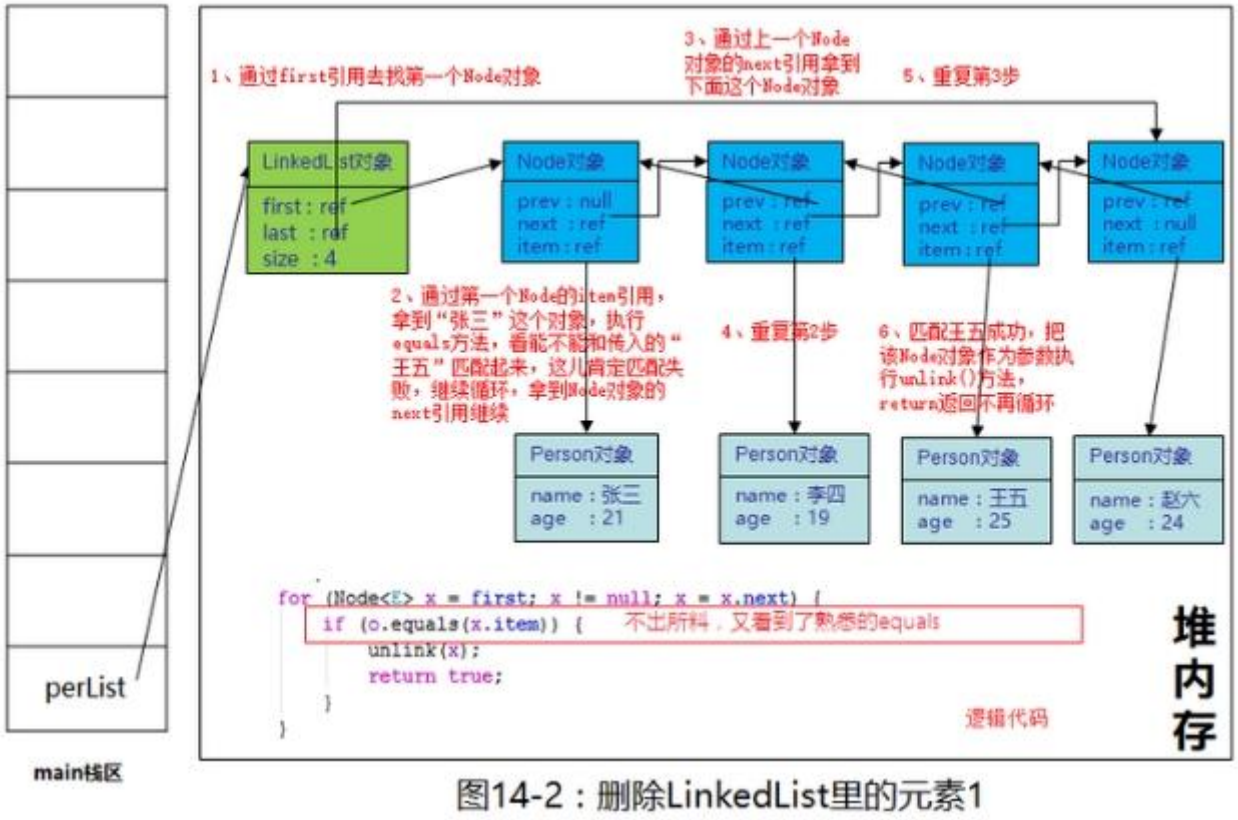


图14-2：删除LinkedList里的元素1

ArrayList(无原子性)和 LinkedList **变安全**：进行读操作时获取读锁，进行增删改操作时获取写锁(用读写锁类 ReentrantReadWriteLock)

4.Vector 也是用数组方式存储，但加了 synchronized 修饰，所以线程安全，但是性能比 ArrayList 差

三大设计模式：创建型(单例模式)，结构型，行为型

单例模式：一个类只有一个实例，并提供一个它的全局访问点(性能高)

懒汉式单例模式：用静态内部实现，使用时才会创建实例对象

```java
//饿汉模式
public class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton() {

    }
    public static Singleton getInstance() {
        return instance;
    }
}
//懒汉模式 线程不安全
public class Singleton {
    private static Singleton instance;
    private Singleton() {
        if(instance == null) {
            instance = new Singleton();
        }
    }
    public static Singleton getInstance() {
        return instance;
    }
}
//懒汉模式 线程安全
public class Singleton {
    private static Singleton instance;
    private Singleton() {
    }
    public static synchronized Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

保证一个类只有一个实例。实现方法：实例存在直接返回，不存在创建实例

工厂模式(创建型)：创建接口，让子类决定实现哪个类
代理模式(结构型)：给其他对象创建代理来控制对一个对象的访问
观察者模式(行为型)：改变一个对象会同时改变其他对象，用于消息队列
生产者消费者：
1. 当队列为空时，消费者线程阻塞，否则唤醒消费者线程
2. 当队列为满时，生产者线程阻塞，否则唤醒生成者线程
因为消费者只从队列里面拿数据，用 take 方法
而生产者只放东西到队列，用 put 方法，这两个方法是独立的

java SE 个人计算机应用 eclipse 里用
java EE 是 java 企业版，用于服务端应用，提供 web 服务，组建模型和管理和通信 API，可以用来实现企业级的面向服务体系结构
java ME—微型版，用于移动产品和车载产品，给设备提供灵活的界面和健壮的安全模式。jvm 把.java 文件编译成字节码文件.class，之后不同的 jvm 都可以解析.class 文件来在不同 OS 上执行

数据库

SQL：结构化查询语言，用于与数据库通讯，
可以创建 1.数据库 2.数据库中的表 3.存储过程 4.图表
DBMS(数据库管理系统)：执行 SQL 查询的软件
SQL vs MySQL：SQL 是一个产品语言，和数据库交流
MySQL 不是语言，是数据库管理系统

inner Join：两张表同时满足的记录
left join：将左表所有查询信息列出来，而右表只列出 On 后条件与左表满足的部分，左表有右表没有的列值为空
right join：将右表所有查询信息列出来，而左表只列出 On 后条件与右表满足的部分，右表有左表没有的列值为空—NULL

MySQL join 原理—nested-loop join

1.简单 nested-loop，取主表(驱动表)每一行找右表每一行去匹配，访问次数多
2.索引 nested-loop 要求非驱动表有索引列，驱动表找到索引后才会回表查询
3.阻塞 nested-loop 把驱动表和 join 相关的列(包括 on 和 select 的列)先缓存到 join buffer 中再去匹配，这样就把第一种 loop 的多次比较合并为一次，效率高

MySQL 执行计划：用 explain 语法来进行查询分析 + navicat 工具

1. SQL 语句执行顺序/执行过程：
from--->join, on--->where--->group by, having--->select, distinct--->order by, limit

2. SQL 一次执行的过程：

1.发 SQL 语句：客户端和 MySQL 服务器建立 TCP 连接后发送一条 SQL 语句给 MySQL 服务器(同一时刻只有一方能发数据，同时，另一方要么完整地接受数据，要么直接断开连接)
2.检查查询缓存：MySQL 检查 SQL 语句是不是 Select 型，是的话去检查查询缓存，缓存命中就返回结果否则去数据库找
3.生成执行计划：MySQL 服务器解析 SQL+预处理(看 SQL 语句有没有错)，然后优化器会预测几个执行计划(指令树)的成本(根据表和索引页数)，选成本最小的那个，不考虑缓存，假设读取都是读磁盘
4.执行：MySQL 遍历指令树，而且调用存储引擎 API 来执行查询
5.返回结果：MySQL 把结果返回给客户端(逐步返回)—结果集每行都会用通信协议封包，然后再通过 TCP 传输

2. **主键**唯一标识一条数据　　不能重复，保证数据完整　一种特殊的唯一索引

外键用来和其他表建立联系　可以重复，是另一个表的主键

3. **索引**(B+树)：一种数据结构，能用它来加快查找数据库数据

没有索引的话，数据按顺序一条条在磁盘上存储(加了主键后，比如自增主键，数据在磁盘变成了平衡树结构，也就是聚簇索引(表排列顺序和索引顺序一致))

索引**优点**：1.创建唯一性索引能保证数据库表中**每一行数据的唯一性**
2.可能可以加快数据检索和**表与表连接速度**

索引**缺点**：1. 会**减慢写入速度+增删改慢**(因为每次写入时都要更新索引)
2.创建索引和维护索引要**消耗时间**，尤其数据量大的时候
3.**占**物理**空间**和数据空间(数据表)—磁盘空间

索引**使用场景/原则**：1.加在经常需要搜索的列上，不要加在写入多和读取少的

列，因为会先访问索引，每次写入时都要更新索引，执行速度变慢
2.数据多+字段值相同的时候用唯一索引(支持 NULL)
3.字段多+没重复的时候用复合索引(不支持 NULL)
4.用唯一性索引(为经常要查询的建索引，限制索引数目)
5.尽量用最左前缀匹配原则，因为如果索引的值很长，比如 text 和 blog 类型的字段，来全文检索的话会浪费时间，如果只检索字段前面的几个字符，能提高索引检索速度
6.where group by order by 的子句要建索引 count max 也要

什么时候不用：1.要取表里所有记录 2.对不是唯一而且重复的字段，比如姓名 3.经常修改和删除的字段 4.记录比较少的表

索引类型：

1.单列索引：只有一个字段的索引 2.**复合索引**：包含两个或以上字段的索引

```
CREATE INDEX 索引名 ON 表名(列名 X, 列名 Y, 列名 Z);
```

建了 3 个索引：
1.单列索引(列 X）2.复合索引(列 X, 列 Y) 3.复合索引(列 X，列 Y，列 Z）
3.唯一索引：表上 一个或者多个字段组合起来建立的索引
字段的值组合起来不可以重复 唯一索引允许有多个 一般只有一个 允许 null
主键：一种特殊的唯一索引 区别：不允许 null 可有可无的

4.聚簇索引—InnoDB 用

1.包含主键索引和对应的数据
2.主键顺序是数据的物理存储顺序，叶子节点是数据节点
3.因为真实数据的物理顺序只有一种，所以一个表最多一个聚簇索引
缺点：如果一个表没有聚簇索引，创建的时候就会对数据重新进行排序，所以它对表的修改速度比较慢，所以不建议在经常更新的列建这个索引
优点：性能好，因为找到第一个索引值，有连续索引值的记录也会在他后面，因为物理上顺序就是在他后面
什么时候用：按照表最常用的 SQL 查询方式来选字段作为聚簇索引
聚簇索引默认是主键，如果表里没有定义主键 InnoDB 会选一个没有 null 值的列代替，如果没有这样的索引，InnoDB 会隐式地定义一个主键作为聚簇索引
聚簇索引 vs 唯一索引：聚簇索引的索引值没有被要求是唯一的，就是在有聚簇索引的列上可以插多个相同的值，这些值在硬盘上的物理排序和在聚簇索的引排序一样

非聚簇索引—MyISAM 用

数据和索引存在不同地方，叶节点仍然是索引节点(存的是主键值)而且是有指向对应数据块的指针
具体怎么用：Mysiam 用 key_buffer 把索引先缓存到内存，当通过索引访问数据的时候，在内存中直接搜索索引，然后通过索引找到磁盘里对应的数据，如果索引不在 key_buffer 命中时，速度就会慢

聚簇 vs 非聚簇区别：聚簇索引能直接查到数据，非聚簇索引先查到主键值，再用主键值查数据

非聚簇索引都是辅助索引，像复合索引、前缀索引、唯一索引一样，辅助索引叶子节点存储的不是行的物理位置，而是主键值

覆盖索引：不用聚簇索引就能直接查到数据，前提是符合最左侧匹配原则

例子：建立索引 createindex index_birthday on user_info(birthday);//查询生日在 1991 年 11 月 1 日出生用户的用户名 select user_name from user_info where birthday ='1991-11-1'

1.通过非聚集索引 index_birthday 查找 birthday 等于 1991-11-1 主键值

2.用主键执行聚集索引找到数据（数据行）存储的位置

3.从得到的真实数据中取得 user_name 字段的值

索引失效：1.列与列对比 2.条件里有 or 3.like 查询以%开头

4.索引列没有限制 not null 5.如果列类型是字符串，没有在条件中把数据用引号引用起来

创建索引：1.普通索引 CREATE INDEX indexName ON mytable(username(length))

ALTER TABLE tbl_name ADD INDEX index_name (column_list)

2.唯一索引

CREATE UNIQUE INDEX indexName ON mytable(username(length))

3.主键索引 添加 PRIMARY KEY

ALTER TABLE `table_name` ADD PRIMARY KEY (`column`)

4.在创建表的时候加

MySQL 和 MongoDB

MySQL：关系型数据库 B+索引和哈希索引(memory 引擎支持，但是重启后数据会丢失)

MongoDB：非关系型数据库(NoSQL), 用的 bson 格式--适合文档存储和查询语句：

```
db.collection.find(); db.collection.distinct("name"); db.collection.find({"age": 22});
db.collection.find({ "field" : { $gte: value } } ); // greater than or equal to : field >= value
db.collection.update( criteria, objNew, upsert, multi )

criteria : update的查询条件，类似sql update查询内where后面的
objNew  : update的对象和一些更新的操作符（如$,$inc...）等，也可以理解为sql update查询内set后面的
upsert  : 这个参数的意思是，如果不存在update的记录，是否插入objNew,true为插入，默认是false，不插入。
multi   : mongodb默认是false,只更新找到的第一条记录，如果这个参数为true,就把按条件查出来多条记录全部更新。

例:
db.test0.update( { "count" : { $gt : 1 } } , { $set : { "test2" : "OK"} } ); 只更新了第一条记录
db.test0.update( { "count" : { $gt : 3 } } , { $set : { "test2" : "OK"} },false,true ); 全更新了
db.test0.update( { "count" : { $gt : 4 } } , { $set : { "test5" : "OK"} },true,false ); 只加进去了第一条
db.test0.update( { "count" : { $gt : 5 } } , { $set : { "test5" : "OK"} },true,true ); 全加进去了
db.test0.update( { "count" : { $gt : 15 } } , { $inc : { "count" : 1} },false,true );全更新了
db.test0.update( { "count" : { $gt : 10 } } , { $inc : { "count" : 1} },false,false );只更新了第一条
```

删除$pull : { field : value } } $表示自己 insert 表示加

MySQL: 坏处：用表存数据在磁盘(读取比较慢) 定义表和字段才能存

好处：有 Join 来进行复杂查询，可以事务处理，可以来保证数据的一致性

NoSQL: 坏处：没有 join，不支持事务，不提供附加功能，比如报表

好处：基于文档设计，把数据放内存，比在磁盘读取快，满足高并发

即使放内存，也比 SQL 简单，因为它是半结构化的数据格式，MySQL 的存储是经过结构化、多范式有很多复杂规则的数据，还原内存结构慢

Redis：缓存数据库，用于存储使用频繁的数据到缓存，读取速度快

而 MySQL 用于持久化地存数据到硬盘，速度慢

Redis 是一个 key-value 存储系统，value 有很多类型而且有 add/remove, push/pop 操作，都是原子性的

数据怎么存放？数据被放在 Mysql 设计的**数据页**上，数据页上数据是**一行行**的，格式是 **compact**，每行都有**行描述**和**指针**去指下一行

行式存储(MySQL) vs 列式存储(Hbase)

1.行式存储倾向于结构固定，列式存储倾向于结构弱化

2.行式存储存储一行数据只要一个主键，列式存储要多个主键

3. 行式存储要维护大量索引和物化视图，所以在时间(处理)和空间(存储)上成本都很高

列式存储只访问查询涉及的列，能很大地降低系统 I/O

数据库事务：DBMS 执行过程中的一个**逻辑单位**，由一个数据库**操作序列**组成

目的：1.提供操作序列从失败到恢复正常的**方法**，同时让数据库在异常情况下仍然能**保持一致性**

2.多个应用程序并发访问数据库的时候**提供一个隔离，**让程序的操作不会干扰到别的操作

4 个**特性**：1.原子性(所有操作要么全都执行成功，要不全部取消)

由重做日志实现

2.一致性(事务开始前和结束后，数据库完整性没有被破坏)

由回滚日志实现

3.隔离性(多个事务并发执行，一个事务的执行不影响其他事务的执行，也就是事务提交之前对其他事务不可见)

由数据库的锁实现

4.持久性(事务提交后，它的结果是永久的) 由重做日志实现

隔离等级：1.读未提交(一个事务可以读其他还没提交的事务的执行结果)

2.不可重复读(一个事务只能读到别的事务已经提交的内容)

3.可重复读(MySQL 的等级)确保一个事务在重复读取数据时候，对重复读取到的数据行的值是不变的

4. 串行化(事务一个个排队执行，只有当事务提交后，其他事务才能从数据库中查看数据的变化) 执行效率很差，可能导致锁竞争

3 种需要阻止的现象：

脏读：一个事务**读到**别的事务写入但还**没提交的数据**

不可重复读：两次查询的内容不一样，因为期间别的事务做修改了

幻读：事务按照之前条件重新查询时，结果集的个数不一致，然后多出来的一行是幻行，也就是读到了别的事务插入的数据

MySQL 解决幻读：1.MVCC(快照读/当前读)：将历史数据**存一份快照**，其他事务**修改数据**是对当前事务**不可见**的

快照：select*from where

当前读：update

2.next-key lock(当前读)：将当前数据行和**上下两行**的间隙**锁定**，查哪些数据行就锁住这些行，保证这个范围内读取的**数据**是**一致**的，但并不完全等于串行化的隔离级别

事务隔离级别	脏读	不可重复读	幻读
读未提交（read-uncommitted）	是	是	是
不可重复读（read-committed）	否	是	是
可重复读（repeatable-read）	否	否	是
串行化（serializable	否	否	否

事务**处理命令**:初始化(start)回滚(rollback)提交(commit)

设保存点(savepoint identifier)

MVCC：每个读操作都会看到一个有一致性的 snapshot，而且可以实现非阻塞的读，允许数据有多个版本，同一时间不同事务看到数据不同

Mysql 的 Innodb 实现 mvcc：给每行加两个字段，分别是这行创建和删除的版本

select：要满足两个条件，Innodb 才会返回当前行的数据：

1.这行的创建版本号小于等于当前版本号，这样保证在 select 操作之前所有的操作已经执行落地

2. 这行的删除版本号大于当前版本或者为空，大于意味着有一个并发事务把这行删除了

insert：把新插的行的创建版本号设为当前系统的版本号

delete：把要删除的行的删除版本号设为当前系统的版本号

update：不执行原地 update，而是转换成 insert + delete。把旧行的删除版本号设为当前版本号，把新行插入然后同时把创建版本号设为当前版本号

同时，写操作（insert、delete 和 update)执行时，要把系统版本号递增

Mysql 事务分类：

1.扁平事务(最简单但最频繁)：begin work 开始到 commit work/rollback work 结束

操作要么都执行要么都回滚

所以是有原子性的应用程序的基本模块

缺点：因为某些地方出错但不会导致所有操作无效的时候还是要回滚之前所有操作，代价太大，因为它只被隐式地设置了一个保存点

2.带有保存点的扁平事务：支持扁平事务的操作+允许事务执行的时候回滚到事务较早的状态，保存点用来通知事务系统应该记住事务当前状态
3.链事务：是保存点事务的变种，在提交事务的时候会释放不需要的对象，把上下文传给下个要开始的事务，这个是一个原子操作
保存点事务的保存点在系统崩溃的时候会丢失，系统恢复后事务要重新执行而不是从最近保存点继续执行
区别：保存点事务能回滚到任意保存点但链事务只能回滚到当前保存点
4.嵌套事务：是一个层次结构框架，有一个顶层事务控制各个层次的子事务
父事务回滚, 子事务也会被回滚—子事务具有 ACI 特性但没有 D 特性
5.分布式事务：是分布式环境下运行的扁平事务
InnoDB 不支持分布式事务，只支持前 4 个，但它支持 XA 事务，通过 XA 来实现分布式事务，而且这个时候隔离等级必须是串行化
Oracle，SQL server 都支持 XA 事务---由 1.资源管理器(提供访问事务的方法)(通常一个数据库就是一个资源管理器)2.事务管理器(协调全局事务里的各个事务)3.应用程序(定义事务边界)

Mysql 日志分类：

1.重做日志：记录事务执行后的状态，来恢复没写进文件的但已经执行成功的事务它更新的数据　保证事务原子性和持久性
2.回滚日志：保存事务发生前的数据，之后可以用于回滚，还提供 mvcc(多版本并发控制下的读)　　保证事务一致性+数据的原子性
重做日志 vs 回滚日志：回滚日志只是把数据从逻辑上恢复到事务开始前的状态而不是从物理页面上操作实现
3.错误日志：记录 mysql 的启动和停止，还有服务器运行过程中发生的错误
4.普通查询日志：记录了服务器接收到每个查询和命令
5.慢查询日志：记录执行时间过长和没有用索引的查询语句

Mysql 存储引擎：

1.InnoDB：是默认的引擎，支持事务但损失效率 有乐观锁和悲观锁
2.MyISAM ：每个 MyISAM 在磁盘上存 3 个文件 fm(存表定义的数据)MYD(存表具体记录的数据) MYI(存索引)　不支持事务但存储更快(允许读写有错误数据的时候用)
3.Memory 把数据存在内存来提高数据的访问速度，每个表和一个 fm 文件关联
用哈希索引

SQL 优化：1.避免全表扫描，首先考虑在 where 和 order by 涉及的列上建索引
2.避免在 where 子句中进行 null 判断和!=<>，or, in, not in
3.尽可能用 varchar 代表 char，这样能节省存储空间
大表优化：1.限定数据范围(禁止那些不带限制数据范围条件的查询语句)2.读写分离(主库负责写，从库负责读)3.垂直分区(根据表的相关性拆分，也就是把数据库列拆分)4.水平分区(表结构不变，把数据分散到不同的表和库，来达到分布式目的)—最好分库因为分表只是解决单一表数据多的问题，表的数据还是在同一台机器上，不能提升 Mysql 并发能力

数据库三范式(为了设计冗余较小、结构合理的数据库)

1. 第一范式：要保证所有**字段**都是**不可分解的原子值**，也就是有原子性-
比如地址列是浙江省杭州市慧芝湖花园，经常访问地址列的话就要把地址这列分成省份 城市 详细地址三列
2. 第二范式：要保证**每列都和主键有关，一个表只能存一种数据**
比如分开订单编号表和商品编号表
3.第三范式：要保证**每列都和主键直接相关**，不是间接相关，然后是用外键连接不同表去得到别的表的信息

数据库的**锁**：悲观锁

类型：1.读锁/共享锁(被锁定的对象只允许被读)事务 A 对一个数据上锁，A 只能读不能写，其他事务只能给这个数据加读锁，除非 A 释放锁
2.写锁/排他锁，事务 A 对一个数据加写锁，事务 A 可以读和改数据但其他事务不能读也不能加锁

表级锁：是系统开销最低但并发性最低的一个锁策略，操作对象是数据库
注意：给表加锁要获得所有涉及到表的锁 LOCK TABLE
InnoDB 和 MySIAM 支持 不会产生死锁
什么时候锁住整个表？ 1.磁盘满的时候 2.insert 时候 3.如果对用 InnoDB 的表使用行锁，被锁的字段不是主键，而且也没有对它建索引的话，行锁锁的是整张表
行级锁：操作的对象是表里的一行，开销大 并发性好 SELECT…LOCK IN
InnoDB 支持
页面锁：开销和加锁时间和锁定粒度介于表锁和行锁之间会，出现死锁，并发度也一般
MVCC(多版本并发控制)：处理并发 开销最大 处理能力最强

死锁：多个资源并发的时候产生。表级锁不会产生死锁
死锁原因：1.事务对资源访问顺序的交替导致(一个用户 A 访问表 A(锁住了表 A)，然后又访问表 B；另外一个用户 B 访问表 B(锁住了表 B)，然后想访问表 A；这样会导致用户 A 因为用户 B 已经锁住了表 B，它就必须等用户 B 释放表 B 才能继续，然后用户 B 要等用户 A 释放表 A 才能继续，这样就产生死锁了) 解决：调整程序的逻辑
2.并发修改同一个记录(用户 A 查一条纪录，然后修改这条纪录；同时用户 B 也想修改这条纪录，这时用户 A 的事务里锁的性质从查询的共享锁想上升到独占锁，但用户 B 里的独占锁因为 A 有共享锁所以必须等 A 释放掉共享锁，而 A 由于 B 的独占锁不能上升独占锁也就不会去释放共享锁，这样就出现了死锁)
解决：乐观锁(实现写-写并发)和悲观锁(保证操作的独占性)
3. 索引不当(2 个情况：1.在事务里执行一条不满足条件的语句导致了全表扫描，把行级锁上升为表级锁，然后多个这样的事务执行后，就很容易产生死锁 2.表中的数据量很大而且索引建的太少或者建的不合适，这样就会经常有全表扫描，这样之后应用系统会越来越慢，最后导致死锁)
解决：SQL 语句中不使用太复杂的关联多表的查询
避免死锁：1.以固定顺序访问表和行 2. 大事务拆小 3.一个事务中一次锁定所有资源 4.降低隔离等级
InnoDB 有自动检查死锁的功能，会自动解决
InnoDB 有三种行锁的算法：

1，Record Lock：单个行上的锁
2，Gap Lock：间隙锁，锁定一个范围，但不包括记录本身
3，Next-Key Lock：1+2，锁定一个范围，而且锁定记录本身。
对于行的查询，都是用这个方法，主要是为了**解决幻读**的问题

B+树：层数一般控制在 3-5 层(IO 次数就是树的高度)，主要为了避免磁盘的读取次数太多，然后它的数据都在叶子节点，并且叶子节点之间用指针连接，这样区间遍历和查询会更快，然后新的值可以插在已经有的节点里，不用改变树的高度，这样就能减少重新平衡和数据迁移的次数，而且中间节点只保存索引不保存数据能容纳更多节点元素

B 树：每个节点都有 key/value, 所以经常访问的节点可能离根节点更近,访问更快

vs B+树：每一层要递归遍历，而且相邻元素可能在内存里不相邻，缓存命中率没有 B+树好，所以 B+树更适合文件系统

缺点：

1、每个节点都有 key 和 data，但是每个页的存储**空间是有限**的，如果 data 比较大的话会导致每个节点存储的 key 数量变小

2、存储的数据量很大的时候会让深度变大，查询时候磁盘 io 次数就多了，这样就会影响查询性能

应用：文件系统和数据库索引

红黑树特点：根节点和叶子节点都是黑色，而且从任一节点到其叶节点黑节点数量一样，二叉树查询复杂度可能是 O(n)--节点都在一边

应用：java 的 treeMap 实现 + IO 多路复用 epoll 的实现 -> 支持快速增删改查

B+/B 树(平衡多路查找树) vs 红黑树：

在同一个节点，B/B+树的高度会远小于红黑树，红黑树 IO 次数就更多了

红黑树的索引时间复杂度稳定在 O(logn)，但是读硬盘读取的时候，会导致读取的数据较多（磁盘读取的基本单位是扇区），根据空间局部性原理，需要读的数据的周围数据也可能读取到，所以使用 B+树更好。

AVL 树 vs 红黑树：控制在 O(logn)，但左右子树高度差不能超过 1，所以经常要调整，但红黑树在插入和删除时不会频繁破坏红黑树的规则，不用频繁调整节点位置

应用：Windows NT 内核

SQL 注入：把用户输入数据作为代码执行

1.**条件**：1.首先用户能控制输入；2.然后是原本程序要执行的代码拼接了用户输入的数据，把数据当代码执行了

2.**类型**：1.没有正确过滤的字符 2.数据库服务器的漏洞 3.盲目 SQL 注入式攻击 4.条件响和条件性差错 5.时间延误

3.**注入点类型**：1. 数字型 2.字符型 3.搜索型

4.**原因**：①查询集处理，转义字符处理，多个提交的处理和错误处理不恰当；②数据库配置不安全

5.**防御**：

1. 用预编译的 SQL 语句，SQL 的语意不会变化，攻击者就不能改变 SQL 的结构了(比如攻击者插入了类似'or '1'='1 的字符串，也只会把这个字符串作为 username 查询)
2.先把 SQL 语句定义在数据库里，从存储过程来防御。(存储过程中可能也存在注入问题，应该尽量避免在存储过程中使用动态 SQL 语句)
3. 限制数据类型，统一数据的格式
4.开发时候尽量用安全函数代替不安全函数，写安全的代码(危险函数 C 里的 system())
5.避免高权限用户直接连接数据库

分布式是指通过网络连接的几个组件，然后通过交换信息协作这样 形成的系统

分布式的 CAP 定理：Consistency（一致性）、 Availability（可用性）、Partition tolerance（分区容错性），最多只能同时三个特性中的两个，三者不可兼得
C：更新操作成功并返回客户端后，所有节点在同一时间的数据完全一致
A：服务一直可用，而且是正常响应时间
P：分布式系统在遇到某节点或网络分区故障的时候，仍然能够对外提供满足一致性或可用性的服务

集群的话是同一种组件的多个实例形成的逻辑上的整体

区别：多个不同组件构成的系统就是分布式系统而不是集群
是集群不是分布式系统的情况的话，比如多个经过负载均衡的 HTTP 服务器，它们之间不会互相通信，如果不带上负载均衡的部分的话，一般不叫做分布式系统

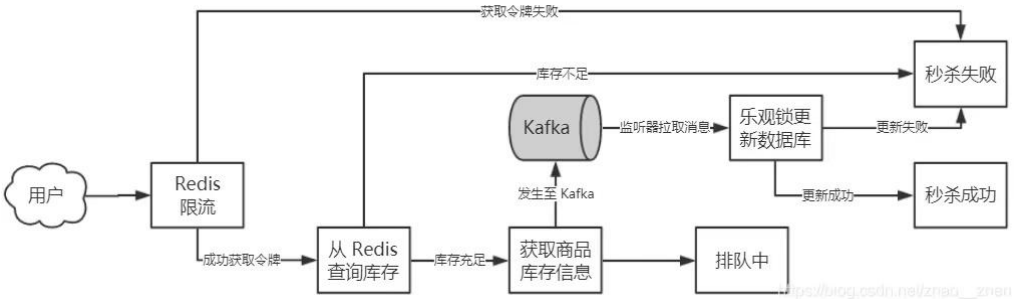
微服务是一种架构模式，它就是把单一应用程序划分成一组小的服务，然后服务之间互相配合来给用户提供价值

每个服务运行在他们独立的进程里，服务和服务之间用比较轻量级的通信机制沟通(通常是基于 HTTP 的 RESTful API)。每个服务都围绕业务来构建，而且他们能被**独立地**部署到生产环境。然后要避免的是统一的管理，对于一个服务，要根据业务上下文，选择合适的语言和工具去构建它。

秒杀系统(将请求尽量拦在系统上游同时对请求进行限流和削峰)

特点：1.高性能：因为有大量并发读和写，所以要支持高并发访问。2.一致性：有限数量的库存同时被很多请求同时来减库存，在大并发更新的过程中要保证数据的准确性。
3.高可用：秒杀时会在一瞬间涌入大量流量，为了避免系统宕机，要做好流量控制。
前端
限流：用验证码来分散用户请求
禁止重复提交：一个用户秒杀后，把提交按钮置灰
动静分离：把静态数据直接缓存到离用户最近的地方，比如浏览器，服务器端的缓存
后端
限流：屏蔽无用的流量，允许少部分流量走后端
削峰：缓冲瞬时流量，尽量让服务器对资源进行平缓处理

异步：将同步请求转化为异步请求提高并发量
利用缓存：把商品信息放缓存中，减少数据库查询
负载均衡：用多个服务器并发处理请求，减少单个服务器压力
Kafka：高吞吐量的分布式发布订阅消息系统



网页爬虫设计：

用例:服务是抓取链接，生成包含搜索词的网页倒排索引，生成页面标题和摘要
用户输入搜索词可以看到相应结果列表
服务有高可用性
假设：搜索流量分布不均，有的搜索词热门有的冷门
用户很快能看到结果，能抓取 10 亿个链接，每月搜索量 1000 亿次

爬虫系统 URL 去重：用数组+hash 函数(求出 URL 的哈希值再对数组长度取模)，但是 hash 冲突时候就会误判(url 长度一样但内容不一样以为存在了),
减少哈希冲突：增长数组长度+用多个 hash 函数判断
布隆过滤器：判断某个元素存在的话可能不存在，但判断某个元素不存在则一定不存在

哈希碰撞：两个不同的输入产生一个相同的 hash 值。

应用：分布式系统里的负载均衡和数据分布

哈希 2 种使用：

1.哈希取模：有 5 个服务器做负载均衡，然后我们就对请求的 ip 和用户 id 用哈希函数，然后把计算出的 hash 值对 5 取模，根据余数来分配到对应的服务器上
缺点：有状态的服务下，新增一个服务器会导致大多数请求要重新映射到别的节点，在 web 负载均衡的情况下，session 会存在每个节点里，如果增删节点会让几乎所有数据要迁移
2.一致性哈希：之前是用节点数量作为除数，一致性哈希的话是用 2^32 作为除数. 也就是 0-2^32 首尾连成一个环。先对服务器节点的 IP 哈希，然后除以 2^32，有请求来了，同样哈希然后除以 2^32, 然后的话是在环上顺时针找到第一个节点，这个节点就负责这个请求
优点：不会因为横向的伸缩导致数据大规模变动
缺点：在节点数量少的时候，会出现分布不均匀的情况(解决方案是在环上加虚

拟节点)

哈希表特点：关键字在表中位置和它之间存在一种确定的关系

哈希冲突：把 key 哈希后的结果作为地址去放键值对的时候(hashmap)，但是地址上已经有别的键值对了

处理哈希冲突/溢出：

1.再散列法：哈希冲突时候，用探测技术在哈希表里得到一个探测序列。按照不同探测序列的形成方法可以分成 3 种：节点规模小时，再散列法节省空间

1.线性探查法：从已存的地址一个个往后找。

缺点：1.处理溢出需要另编程序 2.产生堆聚现象(存入哈希表的记录连成一片)，冲突可能性变大。

2.线性补偿法：探测的步长数要和表的长度互质 这样才能探测表中所有单元

3.随机探测：探测步长是随机数 这样能避免堆聚

2.拉链法：把哈希表变成指针数组，每个单元都是一个头指针，如果键值对有一样的散列地址就插到这个单元的链表里

优点：1.不容易有堆聚现象，平均查找长度短 2.装载因子大，节省空间 3.删节点简单因为不用像再散列法不能直接把节点置位空 因为会截断查找路径

缺点：指针要额外空间，节点规模小时，再散列法节省空间

哈希函数构造方法：

1.数字分析法：知道关键字集合的话，从中选出分布均匀的几位构成哈希地址

2.平均取中法：求关键字的平方值, 然后取平方值的中间几位作为哈希地址

3.分段叠加法：按哈希表地址位数把关键字分成位数相等的几部分然后相加，舍弃最高进位后的结果就是这个关键字的哈希地址

4.除留余数法：哈希函数 key 对小于等于哈希表长度的最大素数取模

版本控制：是记录软件开发过程里文件内容的变化用来查特定版本修订情况的系统

好处：

1. 记录什么人什么时候改了什么内容，每一次文件的改变，文件的版本号都将增加，确保在开发过程中由不同人所编辑的内容都得到更新

2. 方便并行开发，因为可以有效地解决版本的同步以及不同开发者之间的开发通信问题，并行开发中最常见的不同版本软件的(Bug 修正问题也可以通过版本控制中分支和合并的方法解决

版本控制系统：

Git(分布式) 客户端并不只提取最新版本的文件快照，而是把原始的代码仓库完整地镜像下来。这么的话，任何一个服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复

SVN/ CVS(集中式) 一个单一的集中管理的中央服务器，它保存了所有文件的修订版本，然后一起工作的人都可以通过客户端连到这台服务器，拿到最新的文件或者提交更新

内存和缓存：

内存是作为 CPU 和硬盘间的存储支撑，缓存是 cpu 的一部分，在 cpu 里，逻辑上在内存和 CPU 之间。cpu 读数据很快，内存就慢很多。缓存是为了解决 cpu 速度和内存速度差异问题的。内存里被 cpu 访问最频繁的数据和指令被复制到 cpu 的缓存里。在缓存没命中就去内存和磁盘找(数据库在磁盘里，缓存没命中的话数据库会把这些信息更新到缓存)

以下 3 个问题请求少的是影响不大，但请求多会造成服务器宕机，重启也没用
为了查询速度加快用缓存保存数据，让请求直接查缓存而不是数据库，缓存没有的话再查数据库然后写入缓存，缓存是有有效时长的，不然会一直占内存。
缓存雪崩：访问缓存的时候大部分缓存同时过期，请求就会打到数据库，如果请求数量足够大就会把数据库压垮
解决：1.不设置过期时间，缓存更新直接刷新 2.用集群把数据均匀分布在机器上
3.过期时间上加随机值防止缓存几种过期 4.用一些限流机制
缓存击穿：大量请求访问缓存里的一个热点 key，但这个 key 刚好过期了，大量请求就会穿过缓存直接到数据库
解决：1.让热点 key 不设置过期时间 2.设置定时任务把将要过期的 key 刷新
3. 在缓存中没有数据去数据库查询时加上锁，让一个线程去查询数据库以及更新缓存，其他线程等待，这样减小数据库压力
缓存穿透：大量请求访问缓存和数据库都没有的 key
解决：1.在缓存放查询的 key 并且把值设为 null，这样请求就不会打到数据库
2.在请求接口处检查，对于不合法的请求直接返回
3.用布隆过滤器(只有 0 和 1 的 bit 数组)-- 对待过滤的数值求 hash 散列后可以查看这个数组中对应的位置上是否为 1 来进行判断过滤

后端：供客户端的数据请求和响应，客户端跟服务端保持连接、客户端发消息，首先是发给服务端、服务端将消息入库并将消息转发给对方；添加一个好友，是通过客户端操作的，搜索好友即向服务端请求搜索、服务端在“库”里面搜索结果、返回结果给客户端

测试

1. 测试开发是用更全面的技术手段提高测试效率，同时保障产品质量，提升产品交付效率的岗位。现在互联网产业发展很快，一个产品出来，类似产品就会一下就冒出来很多，所以要在市场站住脚产品就要一直试错、迭代和更新。但是修改一个地方可能会牵动全身，所以有时候不得不放弃风险不大的功能模块测试。所以就要测试开发工程师来尽早接入测试，提高测试效率。
2. 为什么选测开：我自己从小就是比较小心谨慎的性格，然后做事喜欢关注一些细节，同时我对测试开发也是很有兴趣的，测试时候可以在设计测试用例上发挥想象力，同时也可以开发测试工具。然后作为我的第一份实习，测试开发可以让我和很多人接触，各方面都能接触到一些，包括产品运转。优秀的侧开是深挖业务，探索不同的测试手段，结合业务去做可以提高效率、解决痛点的工具和平台，去建设质量保障体系。走技术路线，不应该仅仅局限于代码能力，而是解决业务的能力，技术在测试中更多是一种手段
3. 测试和开发的本质区别：软件开发是通过写代码来生成一个软件，也就是从

无到有的过程。而软件测试则是测试一个软件有没有问题，能不能上线，也就是把软件变得更好，起到把关质量的作用。软件开发是有产品产出的，而软件测试则没有，但是这并不影响软件测试的重要性

4. 网页显示空白页如何定位问题：

1、先确保网络连接通畅。

2、查看网络 url 地址是否输入有误。

3、打开控制台查看报错信息。

4、查看接口访问是否有请求。

5、查看路由是否有 path 或者 name 的错误，导致加载了不存在的页面。

5. 测试步骤

单元测试：对最小的模块和重要的控制路径测试，通常是白盒的

集成测试：测试模块接口，采用增量集成

自顶向下: 主模块-隶属于他的模块 自底向上:子模块到满足的顶层模块

系统测试：系统是否满足需求规格，将软件与依赖资源结合，在实际运行下测试(黑盒). 最重要因为能够对所有功能和部件测试,能验证系统是否满足了需求规格的定义

回归测试：修改后测旧例子，验证以前出现但修复的 bug 是否再出现

验收测试：系统开发生命周期方法论一个阶段，系统用户决定是否接受系统

Alpha 测试：用户在受控的开发者场所进行

Beta 测试：软件最终用户在多个场所进行，记录并反馈问题

6. 黑盒白盒测试方法 --设计测试用例的方法

黑盒：是穷举输入测试，已知产品功能来检测功能是否能按照需求规格说明书正常使用，只考虑程序外部结构，不考虑程序内部结构

黑盒方法：

等价类划分：将输入域分成若干部分,从中取代表性数据来测,分有效无效

边界值分析法：选不同等价类的边界值来测

正交实验法：从大量实验点选出适量的、有代表性的点来测

因果图法：描述输入输出的因果关系，用于得到判定表来设计测试用例

状态迁移法：判断状态在给定条件能否产生需要的状态变化

错误推测法：对系统错误操作时处理法的猜测法，用于设计测试用例

白盒：检查程序内部逻辑，每个逻辑路径都要测一次

白盒方法：

静态测试：不运行程序的测试(检查代码,代码质量等分析)

动态测试：通过运行代码找问题，比如接口测试，性能和内存分析

语句覆盖：每个语句至少执行一次

判定覆盖：每个判定分支

条件覆盖：每个判定的每个条件各种可能取值

判定/条件覆盖：同时满足判定+条件

7. 保证用例覆盖度：1.覆盖显性需求：需求文档上的功能

2.覆盖隐形需求 -- 要对行业和竞品进行分析

3.用合适用例设计方法

4.进行用例评审

8. W 模型：测试与开发同步进行，各开发阶段应同步进行验证，优于 v 模型
9. 测试流程：需求测试—概要设计测试—详细设计测试—单元测试—集成测试—系统测试—验收测试
10. 工作：搭建测试环境—写用例—执行用例—写计划报告—bug 表单—跟踪 bug 修改—执行自动化测试—性能测试压力测试等
11. 软件质量：1. 功能特征 2. 可靠特征 3. 易用特征 4. 效率特征 5.可维护特征 6.可移植特征

App 性能测试：1.内存 2. CPU 3.流量 4.电量 5.启动速度 6.滑动切换速度 7. 服务器交互速度

12. 自动化测试：自动化测试与软件开发过程从本质上来讲是一样的,无非是利用自动化测试工具(相当于软件开发工具),经过对测试需求的分析设计出自动化测试用例,从而搭建自动化测试框架,设计与编写自动化脚本,测试脚本的正确性

优点：对回归测试方便；还可以执行一些手工测试难做的测试,比如对大量用户的测试；因为采用脚本，可以使用相当用例,可重复性高还可以立即得到反馈，今早发现 bug

缺点：发现的 bug 比手动化少，依赖测试质量

手工测试：

优点：测试人员有经验和对错误的猜测能力

缺点：依赖测试人员的能力，手工回归测试代价大，容易出错

API 自动化测试：有一个 request collections 按照特定顺序执行，每个测试都是测试某个功能的，Newman(运行和测试 collections), 与公司里持续集成服务一起用

13. bug 周期：new-assigned-open-fixed-pending-reset -reset(再测试)-closed-reopen-pending reject-rejected
14. 测试工具

Jmeter：java 的性能(压力)测试工具

selenium:自动化测试工具

15. 登陆界面测试

测试用例：

功能测试：什么都不输入，输入对的和错的用户名密码的结果，输入内容是否支持特殊字符，密码是否加密显示，大写键盘开启是否有提示，是否记住用户名，牵扯到验证码要看换一个是否有用，登陆成功是否跳转到正确页面

界面测试：页面布局是否合理，

性能测试：打开登录界面要几秒，登录成功跳转新页面不能超时

安全性测试：登陆成功后生成的 cookie 是否会 HTTPonly，用户名密码是否是服务端验证以及是否通过加密的方式发给服务器，错误登录次数(防止暴力破解)，是否支持多用户、多台设备登录

兼容性测试：在不同的浏览器,平台和移动设备上是否显示正常以及功能正常

压力测试：用 jmeter 录制脚本，模拟用户操作得到规定的并发数下(线程数)，多久响应

16. 朋友圈点赞测试：

正常的点赞和取消，收藏和取消收藏，点赞后能否发表评论

点赞的人是否在可见分组
点赞状态有没有更新，共友能不能看见，
网速快慢对其影响
点赞是否按时间排序，头像对应是否正确
是否在消息列表中显示点赞人的昵称，头像等信息

17. 微信发红包测试：

功能：钱数和红包个数只能输入数字，钱=0，塞钱进红包置灰，钱只能小数点后两位，输入钱超过 200, 拼手气红包超过 100 个提醒？最少输入钱不能小于 0.01，红包描述里是否可以输入特殊符号，钱,个数,描述是否支持复制粘贴，是否可以选择和显示红包封面，发的红包能不能领，支付成功有没有回到聊天界面

性能：不同网速发红包抢红包的时间，发收红包的跳转时间

兼容：苹果，安卓都要可以发，电脑端可以抢

界面：有没有错别字，发红包抢红包的界面排版和颜色搭配是否合理

安全：发红包成功和失败，有没有扣钱

18. 微信聊天：

功能：能不能发空白, 正常文字, 超长文字, 特殊符号, 表情,图片,红包,语音
向群，公众号，普通用户和拉黑删除你的用户发
能不能转发 语音转文字 删除 撤回
看历史消息有新消息会不会保留在原来位置
长按文字是否能编辑

性能：不同网速发消息和收消息的时间

兼容：在不同的浏览器,平台和移动设备上是否显示正常以及功能正常

界面：有没有错别字，界面排版和颜色搭配是否合理

19. 对搜索框测试(网页类似)：

功能：输入可查到结果和不可查找结果关键字，输入特殊内容(空格)(等价类划分)，结果显示和排序正常(商家,销量,图片), 模糊搜索，网速慢时搜

性能：不同用户数下的压力表现(响应时间)，承载用户同时使用的极限，常规压力下保持多久稳定运行

易用：交互界面设计：查不到时告知？查到时统计条数并告知？输入框设计风格,位置摆放是否醒目

兼容性：各类浏览器和操作系统下使用，与数据库和监控程序的兼容性

安全：不被 SQL 注入查到加密数据

20. 测试杯子：

功能：水倒到容器一半，安全线，盖子拧紧水倒不出来，烫手验证

性能：使用最大次数和时间，掉地上不易破，保温时间，长时间盛水不漏

界面：外观，大小是否合适，拿着舒服，图案是否掉落

安全：对材质进行毒或细菌验证

兼容性：能够容纳多种液体

21. 测试笔

功能：正常书写？漏笔油？笔帽能不能按下和弹起

性能：能写多久(压力)，是否会褪色

易用性：笔的长短粗细合不合适，笔芯容不容易换

外观：外观是否美观
安全性：笔油是否有毒，笔尖是否容易伤人
适配性：不同的温度气压，不同的纸质和力度书写效果如何

22. 测试直播
    有效等价类：送礼物成功
        观众扣钱，礼物送出，亲密度上升
        主播收到礼物，热榜刷新，界面小喇叭提示，礼物特效和音效
        礼物展示顺序和数量
    无效等价类：送礼物失败
        余额不足，直播结束是否还能送成功
23. 测试电梯
    功能：最基本的运动和按钮功能
    性能：电梯的调度算法，运动速度和耗电量
    压力测试：承重量如何
    兼容测试：是否考虑每个国家电量不一样

前端

1. JS 变量命名：
    1. 0 代表假值，1 代表真值，isShow, isValidating 表示进行中状态，disabled clickable 表示属性状态。
    2. 函数命名：handle...Change.., on..Click 表示事件处理，addUser,fetchToken 表示异步处理，goHome 表示跳转路由，componentDidMount 框架特定方法，getItemById 根据特定属性获取元素
    3.
2. css 框架：Bootstrap 能够结合 JS CSS HTML，提供组件：导航栏，下拉菜单
    与很多浏览器兼容
    还有 Foundation、Materialize
3. 闭包：能够读取其他函数内部变量的函数，是一个定义在函数内部的函数，是连接函数内外部的桥梁，用途/好处：读取函数内部变量和让这些变量存在内存（内部函数有父函数的变量—全局变量，所以内部函数在内存中，父函数也得在内存中），缺点：内存消耗。
4. webpack：前端模块化方案把开发的所有资源看成模块，通过 loader 和 plugins 对资源进行处理，打包成符合生产环境部署的前端资源
5. gulp：强调前端开发的工作流程，我们通过配置一系列的 task，定义 task 处理的事务和执行顺序，让 gulp 执行这些 task，从而构建项目的整个前端开发资源
6. json 只支持 get
7. 实现跨域：window.name + iframe -- 用 iframe 的 src 属于由外域转向本地域
    代理跨域：用一个代理服务器转发数据
8. DOM：文本对象模型，处理可扩展标志语言的标准编程接口。在网页上，组织页面的对象被组织在一个树形结构中
9. addEventListener(event, function, useCapture) event 指事件名，function 指事件触发时执行得函数，useCapture 指事件是否在捕获或冒泡阶段执行

10. fetch 发送 2 次请求，用 fetch 发 post 请求：第一次发 Options 请求，询问服务器是否支持修改的请求头，如果支持，在第二次发真正的请求
11. HTML5 加了 header, footer, nav, section 等标签
12. Java 是面向真正对象编程的语言，JS 是脚本语言
    Java 采用强类型变量(所有变量都要再编译前声明)，JS 变量是弱类型，使用前不用声明
    NaN 是 not a number，是 number 类型但不是数字
13. let 比 var 好因为在函数内部对一个在外面已经声明的变量重新赋值，在函数外面 let 的变量不会变，但 var 会变

const 的对象可以改他的属性 const a = {} a.name = “b” 可行
`${}`里面可以加变量